How Much Can We Micro-Cache Web Pages?

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

ABSTRACT

Browser caches are widely used to improve the performance of Web page loads. Unfortunately, current object-based caching is too coarse-grained to minimize the costs associated with small, localized updates to a Web object. In this paper, we evaluate the benefits if caching were performed at a finer granularity and at different levels (i.e., computed layout and compiled JavaScript). By analyzing Web pages gathered over two years, we find that both layout and code are highly cacheable, suggesting that our proposal can radically reduce time to first paint. We also find that mobile pages are similar to their desktop counterparts in terms of the amount and composition of updates.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Performance of Systems—*Design studies*

Keywords

Caching, Micro-Caching, Web applications, Web pages

1. INTRODUCTION

Over the years, Web pages have become more complex, making Web page loads notoriously slow even though the underlying network has significantly improved. The situation is worse for the mobile scenario, partly due to the lower compute power available on these devices, and partly due to network inefficiencies. Reports have shown that mobile Web applications[1] are losing the market share to their native counterparts [8, 9].

A significant bottleneck of page load times comes from Web page structures, which do little to minimize updates and maximize caching of unmodified content. For example, reloading a page with minor updates requires fetching the entire HTML object, which often triggers a DNS lookup and a TCP connection setup, taking up one third of the page load time [19]. In contrast, a Web page's native counterpart can render right away, interacting with the network for more resources only when necessary. The result is that the native counterpart offers much faster time to first paint [17] (also user-perceived page load time) than the Web page.

[1]In this paper, we use Web pages and Web applications interchangeably since their underlying mechanisms are the same.

IMC'14, November 5–7, 2014, Vancouver, BC, Canada.
Copyright 2014 ACM 978-1-4503-3213-2/14/11 ...$15.00.
http://dx.doi.org/10.1145/2663716.2663739.

We argue that by restructuring Web pages the performance gap between the Web and native applications can be largely eliminated. The first step is to minimize the impact of page updates by employing *micro-caching*. Different from current object-based caching, micro-caching caches any portion of an object at a much finer granularity, which can be an HTML element, a few lines of JavaScript, or a few CSS rules. Implementing micro-caching is possible given that JavaScript now can control almost every aspect of a page and also has access to browser storage.

The second step is to augment browsers with the ability to distinguish between layout, code, and data. Doing so lets us further cache the computed layout and compiled code, substantially reducing computation that is mostly redundant. Note that caching layout, code, and data contributes differently to improving page load times. Caching layout helps the most because loading HTML is always on the critical path of loading a page, which blocks all subsequent object loads [19]. Caching code also helps since loading and evaluating JavaScript are often on the critical path. But, caching data helps little because only a small portion of data fetches are on the critical path. Therefore, our focus is on caching layout and code.

The idea of micro-caching is not entirely new. Delta encoding was proposed over a decade ago, and it advocates that servers transparently compress pages and send differences between versions of a page [3]. Edge Side Includes (ESI) assembles Web pages at the edge to make use of edge caching [4]. We find the idea of micro-caching worth revisiting given that Web pages have become more dynamic than before. Different from treating every bit equally (as in both delta encoding and ESI), our proposal requires applications to make updates explicit and to make distinctions regarding layout, code, and data. As layout and code are likely on the critical path of a page load, caching them separately is key to improving page load times.

As a first step towards micro-caching, this paper studies its effectiveness — how much can we micro-cache Web pages? To this end, we collect snapshots of hundred Web pages over two years, and perform diff-like analysis that identifies updates from one version of a page to another. To infer whether they are from layout, code, or data, we analyze the context of the updates.

While Web pages are traditionally treated as black boxes by measurement studies, we embrace the opposite approach that lets us uncover the redundant bits in Web traffic, and that treats layout, code, and data differently so as to mitigate the bottlenecks of loading modern Web pages. Our main contributions are as follows.

- We propose *micro-caching* that distinguishes portions of a page corresponding to layout, code, and data. We make the case that micro-caching can radically reduce page load times.
- For content-scarce pages, we find that less than 20% (10%) of the HTML page is changed when revisited after a month (day), with updates mostly to code and data, and little to layout. This means that such pages are highly micro-cacheable, especially for layout.
- For content-rich pages, the amount of updates vary across Web pages. In the best (worst) case, 20% (75%) of the HTML page is changed over a month. Most changes are made to data (e.g., links to images, titles) while little is made to layout and code, indicating that layout and code are highly micro-cacheable.
- Mobile pages are similar to desktop counterparts in terms of the amount and composition of updates.
- About half of the object fetches are dynamic and thus the idea of micro-caching is worth revisiting. Unlike CSS and images, most HTML is dynamic. The large amount of dynamic images is also surprising.

2. RELATED WORK

As a key approach to improving page loads, caching has received lots of attention. One related work is delta encoding that advocates compressing HTTP data transparently in the form of differences [3]. Another related work, Edge Side Includes (ESI), assembles Web pages at the edge to exploit edge caching [4]. Unlike delta encoding and ESI that treats every bit equally, we isolate layout and code from data, and cache their evaluated versions separately inside browsers, so as to mitigate the bottlenecks in loading pages, and to remove all the network interactions that block time to first paint in the usual case. Other work focuses solely on object-based caching and treats each object as a black box [14, 21, 16]. We propose micro-caching and are the first to quantitatively and qualitatively study changes in pages at a fine granularity.

There is also a large body of work on measuring other aspects of Web performance. Ihm and Pai [6] presented a longitudinal study of Web traffic, Ager et al. [1] studied Web performance with respect to CDNs, Huang et al. [5] measured the page load times with a slower cellular network, and Wang et al. [18] studied the use of the SPDY protocol to improve page load times. These studies are invaluable for improving our understanding of Web performance.

3. MICRO-CACHING

The case for micro-caching. Web applications are slower than native applications, because Web applications require more steps to launch. Launching a native application requires only executing the code as it has been downloaded beforehand, asking for more data or computation from the server only when necessary (see Figure 1(a)). However, launching a Web application incurs a more complex procedure. It starts with the client asking for the application code from the server, followed by the server running server-side code to generate client-side code and sending it to the client, and finishes with the client running the client-side code, performing layout, and painting (see Figure 1(b)). As

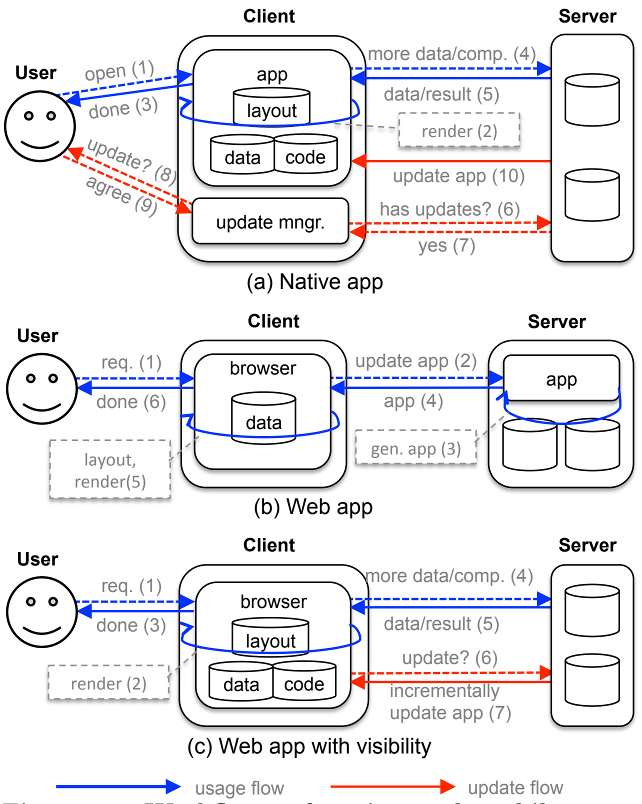


Figure 1: Workflows of native and mobile apps. (solid: data flows; dotted: control flows)

a consequence, Web applications incur much higher time to first paint.

Why do Web applications need more steps to launch? This is because the client side lacks visibility into updates and the model-view-controller (MVC) [10] abstraction of Web applications. The lack of visibility causes three inefficiencies. First, a Web application has to fetch the execution code upon launch even if it hasn't changed, while native applications can explicitly opt in for updates and avoid reloading code. Second, a Web application has to compute the layout from scratch during rendering, while the layout of native applications can be cached. Third, running a Web application always involves JavaScript compilation that happens just in time, while native applications are pre-compiled. Worse for mobile devices, contacting the network sometimes requires a few seconds due to radio interactions, and computation on mobile devices is slower due to the lack of compute power [5, 20]. The result is that Web applications are losing the mobile market to native applications [8, 9].

We argue that, to eliminate these inefficiencies, the client should be provided with enough information — explicit updates and explicit distinctions between layout, code, and data (the MVC abstraction) – both of which are readily available in native applications. The client can then cache not only raw objects but also compiled JavaScript and computed layout, and thus minimize network interactions and computation. We envision the following (as illustrated in Figure 1(c)): both compiled code and computed layout of Web applications are always cached in browsers, incrementally evolving themselves upon updates. When the URL of a Web application is requested, the cached layout is immediately rendered, and the cached code makes the application

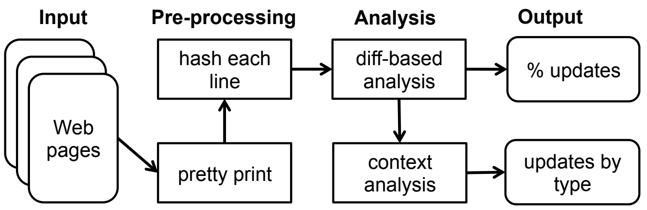


Figure 2: Analysis pipeline.

fully functional including checking the necessity of network interactions, e.g., getting more user data or computation from servers.

This approach removes the entire network interactions that block page loads and portions of computation for the usual case. Therefore, we expect page load times (and time to first paint) to be substantially reduced since our previous study shows that the network time consumes most of the page load time [19].

This approach does not sacrifice any of the existing advantages of Web applications, improves latency, energy consumption, and data usage at the same time, and requires minimal changes to browsers that cache computed layout[2] and compiled code appropriately. Unfortunately, there is no way to provide this visibility automatically without rewriting Web pages because most logics of micro-caching are implemented at the Web page level (e.g., handling versions using HTML5 localStorage). As websites routinely rewrite Web pages, mostly for embracing more efficient architectures (e.g., Groupon and mobile LinkedIn migrated all their pages to node.js [11, 12]), we believe that page rewriting is a viable solution if the benefits are indeed substantial.

Definition of micro-caching. We define micro-caching as caching any portion of an object. Compared to current object-based caching that caches the entire object, micro-caching operates at a much finer granularity. Enabling micro-caching is necessary to provide explicit updates and explicitly distinguish layout, code, and data.

Effectiveness of micro-caching. As a first step towards micro-caching, we study how much we can micro-cache Web pages? We answer this by analyzing the amount of updates from layout, code, and data respectively. This requires us to (i) identify the difference between two versions of a Web page, and (ii) infer whether the difference belongs to data, layout, or code.

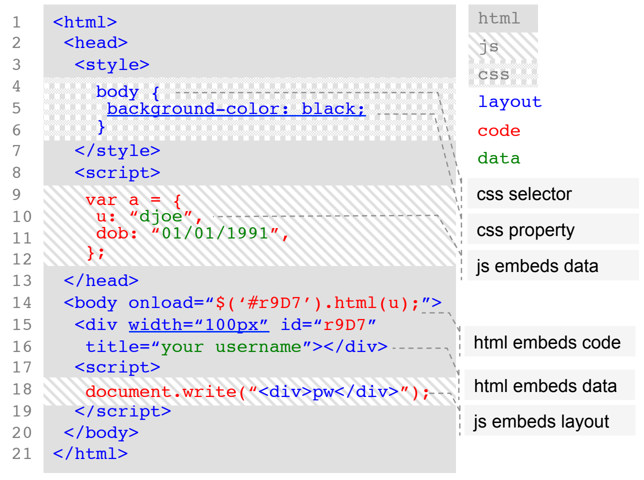
4. METHODOLOGY

Figure 2 shows the pipeline of our analysis that we elaborate below.

4.1 Diff-based analysis

We want to study the difference (or similarity) between two versions of a page. A naive approach is to run a `diff` command that identifies the differences at the granularity of lines. As the definition of a line in Web objects is fragile, we pre-process the pages by pretty printing them using the `js-beautify` library [7]. We use the classic dynamic-programming algorithm that matches two versions of a page to maximize the number of matched lines. A matrix $D_{i,j}$

[2]Modern browsers are able to cache computed layout, but use a different policy to control the lifetime of this cache.



```
1  <html>
2   <head>
3    <style>
4      body {
5       background-color: black;
6      }
7    </style>
8    <script>
9     var a = {
10     u: "djoe",
11     dob: "01/01/1991",
12    };
13  </head>
14  <body onload="$('#r9D7').html(u);">
15   <div width="100px" id="r9D7"
16    title="your username"></div>
17   <script>
18    document.write("<div>pw</div>");
19   </script>
20  </body>
21 </html>
```

Figure 3: Example of inferring context of an HTML document.

is used to denote the number of matched lines between the first i lines of the first page and the first j lines of the second page. If the i-th line of the first page and the j-th line of the second page are matched, $D_{i,j} = D_{i-1,j-1} + 1$; otherwise, $D_{i,j} = max\{D_{i-1,j}, D_{i,j-1}\}$. We start with $D_{i,0} = 0$ and $D_{0,j} = 0$, and increase the indice until we reach the last lines of both pages. This algorithm incurs a $O(n^2)$ time complexity where n is the number of lines of a page. As some lines can be unexpectedly long and slow down the algorithm, we accelerate this step by calculating an `md5` hash of each line and matching the hash values. Our algorithm outputs the number of unmatched lines, bytes, and unmatched content.

We use *similarity*, defined as one minus the fraction of difference in bytes using the above algorithm, to characterize the amount of updates eliminated by micro-caching. *Similarity* provides a lower bound on estimated savings since unmatched lines of two pages are not entirely different.

4.2 Context analysis

We first infer whether a line is one of HTML markup, CSS, and JavaScript, and then infer whether a string belongs to layout, code, or data.

Inferring HTML/CSS/JS. Inferring HTML is straightforward; we infer a line as HTML if it is quoted by `<>` (pretty printing helps here). However, differentiating between CSS and JavaScript is non-trivial; for example, a CSS property is similar to a property declaration of an object in JavaScript (e.g., Line 5 and 10 in Figure 3). We notice the slight difference that CSS property (JavaScript object property) ends with a semi-colon (comma), and use it to distinguish between CSS and JavaScript. When we are unable to distinguish by looking at the line itself, we search backwards and forwards until hitting a line that allows for inference.

Inferring layout/code/data. Before this inference, we further identify the modified and added content at the granularity of strings. The diff-based analysis gives pairs of unmatched chunks (a few lines) between two pages, indicating that a chunk on the first page is modified into another chunk on the second page. For example, insertions (removals) incur an empty chunk on the first (second) page. Here, we first break up a chunk into strings that are separated by either spaces or the newline character (pretty printing also helps

Page	Content	Country	Category
google.com	scarce	US	Search
baidu.com	scarce	China	Search
wikipedia.org	scarce	–	Information
amazon.com	rich	US	E-Commerce
taobao.com	rich	China	E-Commerce
youtube.com	rich	US	Video
yahoo.co.jp	rich	Japan	News

Table 1: Seven pages for extensive analysis.

here). For each pair of unmatched chunks, we apply the diff-based analysis above to pinpoint the unmatched strings.

Inferring from CSS is trivial since all of CSS maps to layout, but inferring from HTML and JavaScript are not trivial. For HTML, attributes can be layout when they are like CSS properties (e.g., `width`), can be code when they start with `on` (e.g., `onload`), and can be data when they are one of `value`, `name`, and so forth. JavaScript is code by default, but can embed data (e.g., username), and can embed code that is quoted by `document.write()` or `eval()`. To infer from HTML, we pre-classify all known attributes into the appropriate layout, code, or data categories. We are unable to classify self-defined attributes. To infer from JavaScript, we identify strings that are quoted by single or double quotes as data. We do not encounter code that starts with `document.write` or `eval`, and therefore we do not have to handle that case[3]. Figure 3 illustrates the context inference of an HTML document.

4.3 Datasets

We collect our datasets using a measurement node at the University of Washington, starting from April 2012. Our datasets span two years. We extensively take snapshots of top twenty Alexa [2] pages every hour, which is more frequent than page updates. We also take snapshots of top one hundred Web pages every day. To study mobile pages, we collect a month-long dataset of top twenty mobile pages every hour. To take a snapshot of a page, we use both PhantomJS [13] to collect HTTP headers and Web page metadata in the HAR format and `wget` to collect the content of Web objects.

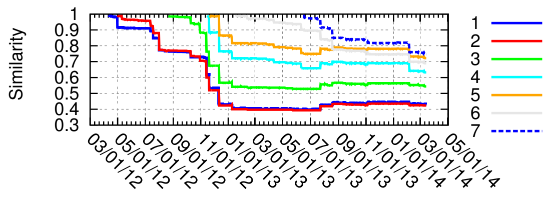
5. RESULTS

We study the benefits of micro-caching and shed light on existing caching schemes.

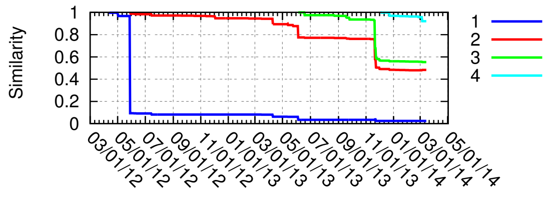
5.1 Benefits of micro-caching

Filtering Web pages. The set of measured pages are either content-rich pages that provide personalized or up-to-date content (e.g., news, video, e-commerce), or content-scarce pages that provide a single service (e.g., search). In our study, we exclude pages that require logins to provide content (e.g., social network) since these pages without logins are not representative of their real usage. We also exclude pages that are similar to a chosen page (e.g., `google.de` is similar to `google.com`). By excluding such pages, we focus on seven of the top twenty pages for intensive analysis. These pages (shown in Table 1) contain both content-rich

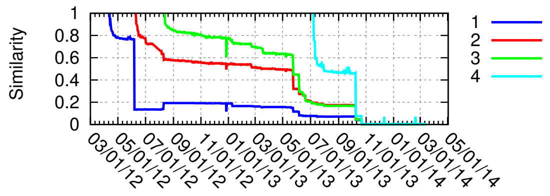
[3]`eval` is considered a bad practice, but was found on half of the top 10,000 pages [15]. We do not find it on our set of pages, likely because the very top pages are optimized.



(a) google.com



(b) wikipedia.org



(c) taobao.com

Figure 5: Similarity over time that reflects how websites do updates. Each curve represents similarity between a fixed version and versions after.

and content-scarce pages and span three countries and five categories. We believe that intensively studying a small set of top pages is valuable, because they are often the most dynamic and are thus hardest to be micro-cached.

Estimating benefits. A direct way of estimating benefits of micro-caching would be comparing the page load times before and after micro-caching is applied. However, such comparison is both inaccurate and unfair. It is hard to accurately estimate the page load time benefits without implementing micro-caching since page load times depend on a large number of factors. This includes network parameters, compute power of the browser device, and the dependency model of page load activities. A slight modification to the Web page can result in very different page load times. To enable micro-caching, we propose to modify the entire Web pages so that pages are loaded directly from the browser cache most of the time and are updated asynchronously only when necessary. Comparing page load times of the modified page and the original page would be unfair.

Here, we qualitatively estimate the benefits. Our previous study informs that the network time consumes most of the page load times [19]. For example, contacting remote servers often triggers DNS lookup, TCP connection setup, and SSL handshakes when HTTPS is used; the cascades of latencies make time to first paint slow. Our proposal removes the entire network interactions and portions of computation that block page loads for the usual case. Thus, we expect page load times to be largely reduced.

Instead of quantifying the benefits of page load times, we focus on studying the impact from Web page updates. The

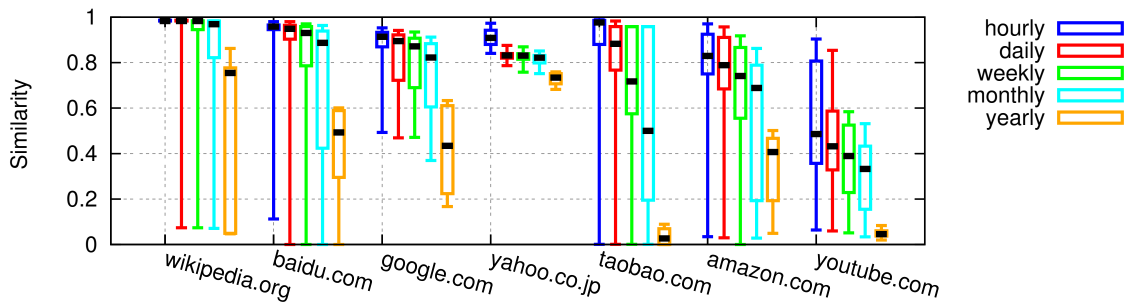


Figure 4: Similarity between two versions of pages by varying access interval. For a fixed interval, we vary the time of accessing the first version and obtain a list of similarities. The candle sticks show minimum, 10-percentile, median, 90-percentile, and maximum.

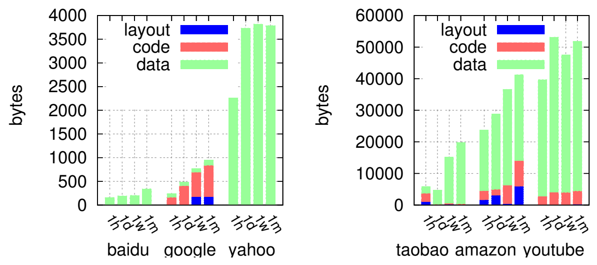


Figure 6: Updated bytes broken down by layout, code, and data. `wikipedia.org` is excluded as little is updated.

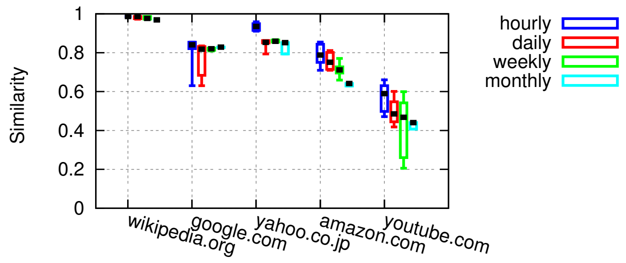


Figure 7: Similarity between two versions of mobile pages by varying access frequency.

less and less often a Web page is updated, the less and less often an asynchronous network fetch needs to be issued, and the more the page can be micro-cached.

5.1.1 Longitudinal study

We report on the two-year desktop dataset.

How much is updated. We first look at the amount of updates when visiting a page after a given interval. Here we use *similarity* that indicates an upper bound on the amount of updates. Figure 4 shows the results. For content-scarce pages (left three), less than 20% (10%) of the HTML page is changed when revisited after a month (day), meaning that such pages are highly micro-cacheable. Content-rich pages (right four) have high variance for micro-cacheability: `youtube.com` updates significantly every hour, while `yahoo.co.jp` updates less than 20% even over a month.

How often pages are updated. We further look at how often pages are updated by obtaining similarity over time in Figure 5. Here, we focus on major updates that require reloading a large portion of the page. Updates are indicated by sharp drops in Figure 5 – the more a page is updated, the sharper the drop is. We find that `google.com` updates every few weeks. `wikipedia.org` updates less often; we see one major update at the end of May 2012 and one at the end of November 2013. Unlike content-scarce pages, `taobao.com` updates incrementally between two major updates. These incremental updates are mostly data, but little layout or code. We also plot the graphs for other pages, which we do not show here due to space limits. They together show that major updates are rare and the amount of updates in layout and code between two major updates are moderate, suggesting that micro-caching is practical to mitigate page load bottlenecks.

What is updated. We break down the updated bytes into layout, code, and data. Figure 6 shows that most updates are made to data; `baidu.com` and `yahoo.com` update only data. Updates to data are mostly through HTML attributes (e.g., `href`, `title`) and inner HTML content, while some updates to data are made through JavaScript. Conversely, little update is made to layout; only half of the pages update layout, by a lesser amount. Also, little update is made to code. These results together suggest that layout and code are highly micro-cacheable for both content-scarce and content-rich pages.

To be informative, we manually go through some of the updated content. For example in `google.com`, we find that many updates are random strings/numbers that are generated by servers. Except for security considerations, random string/number generation can be moved to the client side so as to minimize updates. We also find that some pages change CSS that has no visible impact. This CSS is likely being loaded speculatively that will be used when users perform an action. This kind of CSS can be loaded in the background without impairing the cacheability of layout. Some other CSS shows significant visual impact. Because we can cache the previous layout, incremental layout incurs minimal computation.

5.1.2 Mobile pages

We report on the month-long mobile dataset. We exclude two pages that provide different functionalities than their desktop counterparts. Figure 7 shows the mobile counterparts of Figure 4. Clearly, the amount of updates in mobile Web pages are similar to their desktop counterparts. The variance is smaller here because the time of measurements

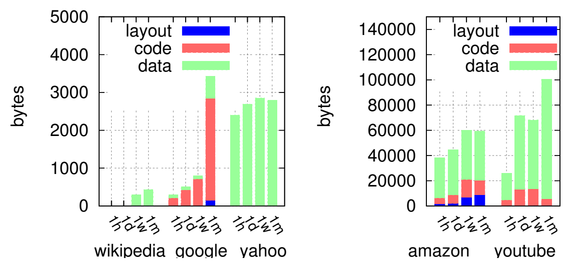


Figure 8: Updated bytes of mobile pages broken down by layout, code, and data.

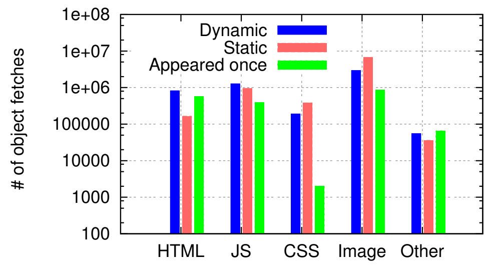


Figure 9: Breakdown the number of static, dynamic, and once objects by mime type.

is shorter. Figure 8 shows the mobile counterpart of Figure 6. They are alike except for some random spikes in Figure 8. We manually look through updates in mobile pages and desktop pages and find that updates in code are similar, indicating that they are likely to share the same code base. However, the format of updated data is different, possibly because of delivering content on a smaller screen.

5.1.3 *Practices that undermine micro-caching*

We summarize common practices suggested by our measurements that undermine the effectiveness of micro-caching.

- **JavaScript obfuscation.** Websites often obfuscate client-side JavaScript to reduce readability which, however, also reduces micro-cacheability. We find variables in two versions of `google.com` performing the same logic while using different names.
- **CSS reordering.** The order of CSS rules does not matter most of the time and we find some websites reorder their CSS rules. Using consistent ordering would help micro-caching.
- **CSS abbreviation.** We find equivalent CSS rules in different versions of a page. For example, `body,a{rule}` is unfolded to `body{rule} a{rule}`, and `border-top`, `border-bottom`, `border-left`, and `border-right` are condensed into `border`. Using the same level of abbreviation consistently would help micro-caching.
- **Object sharding.** In the HTTP/1.1 era, websites shard JavaScript and CSS objects to exploit concurrent TCP connections. Because sharded objects use random URLs as their keys, they hurt caching.

5.2 Dynamics of Web objects

Our measurements also let us analyze the dynamics at the granularity of a Web object. Static objects can be cached infinitely given enough disk space, while dynamic objects need to evict their cache before updates. Studying the dynamics at the granularity of a Web object sheds light on implementing expiration of object-based caching. Here, we consider all pages we collected.

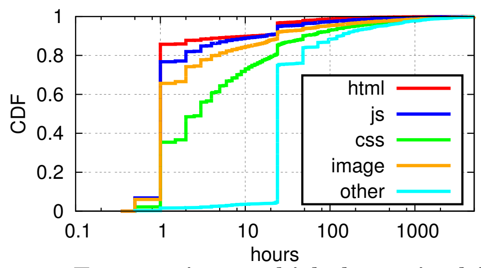


Figure 10: Frequencies at which dynamic objects are updated (breakdown by mime type).

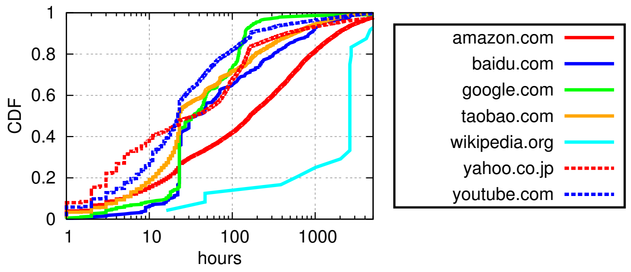


Figure 11: Lifetimes of static objects (breakdowns by popular domain).

We find that about half of the object fetches are dynamic, meaning that the idea of micro-caching is worth revisiting. Figure 9 shows the amount of static and dynamic objects respectively broken down by MIME type. HTML and JavaScript are more dynamic while CSS and images are more static. Images are expected to be identified by a unique URL, but we are surprised to learn that a significant amount of images are dynamic.

Figure 10 shows the frequencies at which dynamic objects are updated. Most dynamic HTML, JavaScript, and images are changing all the time (1 hour is the unit of measurements), likely being backed by server-side scripts. In contrast, less than 40% of CSS is changing all the time. For all kinds of dynamic objects, over 75% of them change within a day, meaning that cache expiration for dynamic objects should be mostly set to just a few hours.

Figure 11 shows the lifetimes of static objects. The median lifetime of static objects is about two days. However, there is high variance in lifetimes regarding different websites. Over 75% of objects in `wikipedia.org` have a lifetime of more than 1,000 hours, suggesting that most static objects in `wikipedia.org` should be cached. But half of the static objects on most other pages have a lifetime of less than a day, suggesting that caching these objects for more than one day is likely to waste disk space.

6. CONCLUSION

This paper proposes to separately cache layout, code, and data at a fine granularity inside browsers to mitigate the bottleneck of loading modern Web pages. By analyzing two years of measurements of Web pages, we find that layout and code that block subsequent object loads are highly cacheable.

Acknowledgements

We thank our shepherd, Andreas Haeberlen, and the anonymous reviewers for their feedback.

7. REFERENCES

[1] B. Ager, W. Muhlbauer, G. Smaragdakis, and S. Uhlig. Web Content Cartography. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011.*

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

[3] Delta encoding in HTTP. `http://tools.ietf.org/html/draft-mogul-http-delta-07`.

[4] ESI Language Specification 1.0. `http://www.w3.org/TR/esi-lang`.

[5] J. Huang, Q. Xu, B. Tiwana, Z. M. Mao, M. Zhang, and P. Bahl. Anatomizing application performance differences on smartphones. In *Proc. of the international conference on Mobile systems, applications, and services (Mobisys), 2010.*

[6] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *Proc. of the SIGCOMM conference on Internet Measurement Conference (IMC), 2011.*

[7] JS Beautify. `http://jsbeautifier.org/`.

[8] The mobile web is still losing out to native apps, by more than 6 to 1. `http://venturebeat.com/2014/04/01/the-mobile-web-is-still-losing-out-to-native-apps-six-years-into-the-mobile-revolution/`.

[9] Apps Solidify Leadership Six Years into the Mobile Revolution. `http://www.flurry.com/bid/109749/Apps-Solidify-Leadership-Six-Years-into-the-Mobile-Revolution#.U2bgha2SzOQ`.

[10] Model-view-controller. `http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`.

[11] Need for Speed: How Groupon Migrated to Node.js. `http://www.datacenterknowledge.com/archives/2013/12/06/need-speed-groupon-migrated-node-js/`.

[12] Many high-profile companies – node.js. `http://nodejs.org/industry/`.

[13] PhantomJS. `http://phantomjs.org/`.

[14] F. Qian, K. S. Quah, J. Huang, J. Erman, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck. Web Caching on Smartphones: Ideal vs. Reality. In *Proc. of the ACM Mobisys, 2012.*

[15] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval that Men Do. In *Proc. of the 25th European Conference on Object-Oriented Programming (ECOOP), 2011.*

[16] S. Sundaresan, N. Feamster, R. Teixeira, and N. Magharei. Measuring and Mitigating Web Performance Bottlenecks in Broadband Access Networks. In *Proc. of the ACM Sigcomm Internet Measurement Conference (IMC), 2013.*

[17] Transaction Perspective: User Experience Metrics. `http://www.keynote.com/products/web_performance/performance_measurement/user-experience-metrics.html`.

[18] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Accelerating the Mobile Web with Selective Offloading. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2014.*

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

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

[21] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proc. of the international conference on World Wide Web (WWW), 2012.*