Rethinking Client-Side Caching for the Mobile Web

Ayush Goel
goelayu@umich.edu
University of Michigan

Vaspol Ruamviboonsuk
vaspol@umich.edu
University of Michigan

Ravi Netravali
ravi@cs.ucla.edu
UCLA

Harsha V. Madhyastha
harshavm@umich.edu
University of Michigan

ABSTRACT

Mobile web browsing remains slow despite many efforts to accelerate page loads. Like others, we find that client-side computation (in particular, JavaScript execution) is a key culprit. Prior solutions to mitigate computation overheads, however, suffer from security, privacy, and deployability issues, hindering their adoption.

To sidestep these issues, we propose a browser-based solution in which every client reuses identical computations from its prior page loads. Our analysis across roughly 230 pages reveals that, even on a modern smartphone, such an approach could reduce client-side computation by a median of 49% on pages which are most in need of such optimizations.

CCS CONCEPTS

• **Information systems** → **Web applications**; **Browsers**.

KEYWORDS

Mobile web, client-side computation, JavaScript caching

ACM Reference Format:
Ayush Goel, Vaspol Ruamviboonsuk, Ravi Netravali, and Harsha V. Madhyastha. 2021. Rethinking Client-Side Caching for the Mobile Web. In *The 22nd International Workshop on Mobile Computing Systems and Applications (HotMobile '21), February 24–26, 2021, Virtual, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3446382.3448664

1 INTRODUCTION

Recent years have witnessed significant growth in the amount of web traffic generated through mobile browsing [8]. Unfortunately, mobile web performance has not kept up with this rapid rise in popularity. Mobile page loads in the wild are often much slower than what users can tolerate [23], with many pages requiring more than 7 seconds to fully render [5].

A key contributor to slow mobile page loads is client-side computation—in particular, JavaScript execution—as seen in our measurements (§2) and in prior studies [30, 41]. Given the importance of fast page loads for both user satisfaction [10] and content provider revenue [1], much effort has been expended to alleviate this bottleneck by reducing the work that mobile devices must do to load pages. However, despite their promising results, existing solutions have (fundamental) practical drawbacks that have hindered adoption.

- Offloading computation tasks in page loads to well-provisioned proxy servers, which ship back computation *results* for clients to apply locally [15, 38, 47], poses numerous security and scalability challenges. Clients must trust proxies to preserve the integrity of HTTPS objects, and they must share (potentially private) HTTP Cookies with proxies in order to support personalization. In addition, proxies require non-trivial amounts of resources to support large numbers of mobile clients [42].
- Having origin web servers return post-processed versions of their pages which elide intermediate computations [33] results in fragile content alternations that may break page functionality [14]. For example, pages may adapt execution based on client-side state (e.g., localStorage); servers are inherently unaware of this state while generating post-processed pages, and thus risk violating page correctness. Moreover, like proxy-based solutions, this approach places undue burden on web servers to generate optimized versions for the large number of pages they serve (including versions personalized to individual users).

We argue that the key to easing deployability is to shift the focus to solutions which only require client-side changes. Doing so sidesteps the security, privacy, and correctness concerns discussed above. Moreover, only a handful of browsers need to be updated for most users to benefit [2].

As a first step towards this vision, in this paper, we ask: *how much web computation can be eliminated by a purely client-side solution?* To answer this question, we propose a rethink of the functionality of client browser caches. While client-side caching has been a staple optimization in page loads for decades, browsers have used their caches only to eliminate *network fetches*; recent caching proposals for improved hit rates share the same focus [34, 46]. In contrast, we propose that browser caches be extended to enable *reuse of computations* from prior page loads.

The idea of computation reuse, commonly known as computation memoization, dates all the way back to late 1960s [28] when the idea of a function *"remembering"* results corresponding to any set of specific inputs was first introduced. Memoization has found wide applicability in language compilers [36, 44, 45] as well as other domains such as image search [25], image rendering [17] and data center computing [16, 24, 26]. In this paper, we study the potential of such an approach in the context of web page loads and make contributions along the following dimensions:

(1) **Granularity.** The granularity at which computation is cached can have a significant impact on potential benefits, and must be amenable to the fact that cache entries may be from page loads performed several minutes or even hours ago. For example, we find that 11% of JavaScript code on the landing page of

HotMobile '21, February 24–26, 2021, Virtual, United Kingdom

ACM ISBN 978-1-4503-8323-3/21/02…$15.00
https://doi.org/10.1145/3446382.3448664

the median Alexa top 500 site changes each hour, thereby precluding computation reuse at a page level [33, 38]. Instead, we propose finer-grained computation caching at the granularity of JavaScript functions. Our proposal is rooted in our finding that 96% of JavaScript code is housed inside JavaScript functions on the median page.

(2) **Efficacy.** To measure the potential benefits of our proposal, we developed an automated JavaScript tracing tool that dynamically tracks all accesses to page state made by each function invocation in a page load; this information is required to determine the reusability of computation from prior page loads. We experiment with a state-of-the-art phone (Google Pixel 2) and over 230 pages (landing pages of top Alexa sites and random sites from DMOZ [4]). For the subset of these pages which require clients to perform over 3 seconds of computation ("high-compute pages"), we estimate that client-side reuse of JavaScript executions can eliminate 49% of client-side computation on the median page.

(3) **Practicality.** Finally, we sketch the design of a browser-based system that performs computation caching. We outline the practical challenges of such a system, which largely revolve around the high state tracking and cache management overheads. To alleviate those overheads without sacrificing substantial reuse opportunities, our key finding is that 80% of total JavaScript execution time is accounted for by 27% of functions on the median high-compute page. This allows for the system to target only a small fraction of functions while reaping most of the potential computation caching benefits.

2 MOTIVATION

We begin by presenting a range of measurements to illustrate the large (negative) impact that client-side computation has on overall page load times (PLTs). Our experiments use a modern smartphone (Google Pixel 2[1]) with Google Chrome (v73), and consider landing pages from the Alexa top 1000 sites; these pages are more likely to incorporate recommended best practices for enabling fast page loads.

Mobile web page loads often have very high compute. We record each page and then load it within the Mahimahi replay environment [35] over an emulated 4G network [7]; emulation was done using Chrome's network shaping feature. We focus our analysis on the 223 pages which experienced PLTs greater than 3 seconds (the *"Shaped, all cores"* line in Figure 1), since these loads are slower than user tolerance levels [23].

Since page loads consist of fetching resources over the network *and* processing those resources to display functional content, the observed load times could be high due to network or computational delays. To distinguish between these two factors, we loaded the 223 slow landing pages again using Mahimahi, but this time with an unshaped network. We note that this represents the best case performance for prior (complementary) web optimizations that target network delays [18, 32, 43, 48]. As shown in the *"Unshaped, all cores"* line of Figure 1, 39% of these pages (i.e., 86 pages) continue to experience load times greater than 3 seconds, despite the lack of network delays; we call these "high-compute" pages. While client-side computation may not always be the primary bottleneck when

[1]We believe a more recent version of Google Pixel (e.g., Pixel 4) would show some, albeit limited, improvements in the total client-side computation time due to a slightly higher CPU clock speed [20].

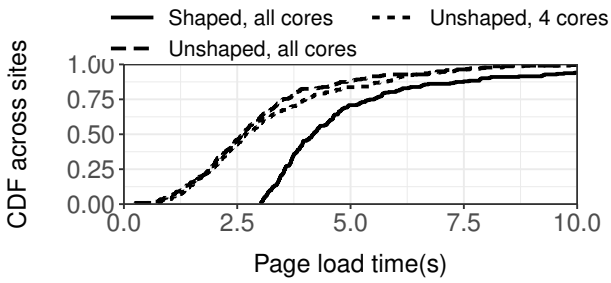


Figure 1: Page load times, with or without network delays (shaped vs. unshaped) and when using all 8 CPU cores or only 4 of them. Results are for 223 landing pages with load times over 3 seconds with a shaped network.

these pages are loaded in the wild (i.e., network fetches may constitute the critical path), these results show that compute delays alone would slow down many web pages beyond user tolerance levels.

Our findings, while in line with recent work [30, 41], are in stark contrast to observations made by earlier studies. For example, a decade ago, Wang et al. [48] found that high network latency and the serialization of network requests in page loads are the key contributors to poor mobile web performance. Since then, the mobile web landscape has changed significantly in three ways. First, due to a 680% increase over the last 10 years in the number of bytes of JavaScript included on the median mobile page [9], the amount of client-side computation as part of web page loads has dramatically increased. Second, the quality of mobile networks has improved greatly, e.g., over the last 10 years, on the average mobile connection globally, bandwidth has increased from 1MBps to 19MBps and RTT has decreased from 700ms to 65ms [11–13]. Lastly, the increased adoption of HTTP/2 has reduced the serialization of network requests; while HTTP/2 did not exist a decade ago, the fraction of requests on the median page that are served over HTTP/2 is now up to 67% [6].

Compute will continue to slow down page loads. Improvements in CPU performance generally come from increase in either the number of cores or the clock speed of each core. However, with mobile devices, improvements have been largely due to the former, with clock speeds increasing at a far slower rate, e.g., CPU clock speed on the Samsung Galaxy S series increased from 1.9GHz in 2013 to 2.73GHz in 2019; the number of CPU cores doubled during that time (from 4 to 8).

Unfortunately, this trend of increased cores provides little benefit to the mobile page load process. Web browsers are more dependent on clock speed than the number of cores [21] because they are unable to fully take advantage of all available CPU cores (described more below). Indeed, Figure 1's *"Unshaped, 4 cores"* line shows that PLTs are largely unchanged even when we disable 4 out of 8 CPU cores on the Pixel 2.

Cellular networks, on the other hand, are projected to continue to get significantly faster [3]. Given these trends, as well as the energy restrictions that hinder CPU speeds on mobile devices [39], client-side computation will likely continue to significantly contribute to load times.

JavaScript execution dominates computation delays. Computation delays in page loads stem from numerous tasks that browsers must perform, such as parsing and evaluating objects like HTML,

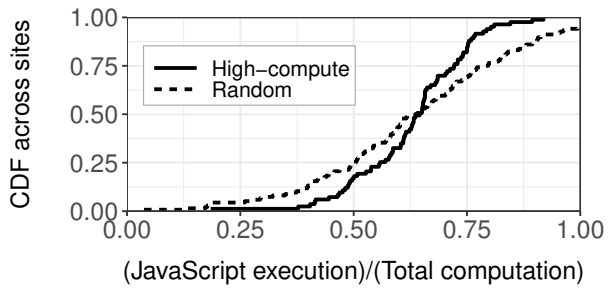
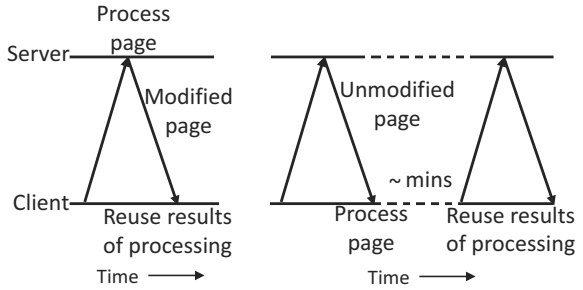


Figure 2: Fraction of browser computation accounted for by JavaScript execution. Results are for 200 random landing pages in the Alexa top 1000 and 86 high-compute pages.



(a) Server-side technique (b) Client-side technique

Figure 3: (a) Server-side acceleration techniques send a processed page to the client. (b) Client-side acceleration requires initial web page loads to populate the computation cache, and then subsequent page loads to utilize this cache.

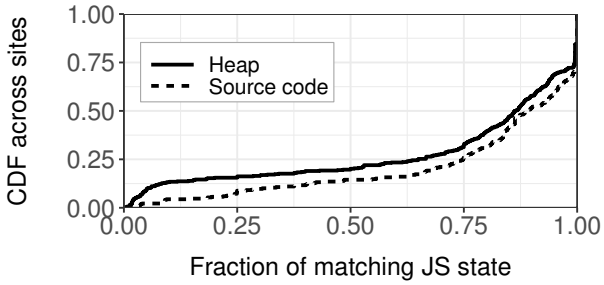


Figure 4: Fraction of JavaScript that matches across two loads of the same page one hour apart.

CSS and JavaScript, and rendering content to the screen. Furthermore, the JavaScript engine in the browser spends time compiling and executing JavaScript, and doing garbage collection. We analyzed the loads of each page in our corpus to identify which of these compute tasks browsers spend the most time on. We consider two sets of pages: 200 random pages from the Alexa top 1000 sites, and the 86 high-compute pages from above.

We find that JavaScript execution is the primary contributor to browser computation delays in page loads. In particular, Figure 2 shows that JavaScript execution accounts for 64% and 65% of overall computation time for the median page in the two sets of pages, respectively.

This explains why page load performance does not benefit much from more cores, as we saw above. JavaScript execution in browsers is single-threaded and non-preemptive for each frame in a web page [40]. While this single-threaded model greatly simplifies web page development, it does so at the cost of degraded performance and resource utilization.

3 CLIENT-SIDE COMPUTATION REUSE

To overcome the practical limitations of prior systems (§1), we advocate for a purely client-driven approach to reduce client-side computation in mobile page loads. Rather than having clients reuse the results of *server-side* or *proxy-side* page load processing, we envision each client reusing computations from *its own* page loads from the past. More specifically, like how web browsers cache objects to exploit temporal locality in a client's page loads [49] and eliminate redundant network fetches, we propose that browsers also reuse JavaScript executions from prior page loads.

3.1 Need for Fine-Grained Computation Reuse

Although conceptually straightforward, our proposal necessitates a fundamentally new approach for how computation is reused. The primary difference between our vision and existing proxy-/server-side approaches is that of timing (Figure 3). In existing solutions, a proxy/server loads a page in response to a client request and returns a compute-optimized version, which the client applies a few seconds later. In this workflow, clients download a single post-processed object that reflects *all* of the state in the latest version of the page.

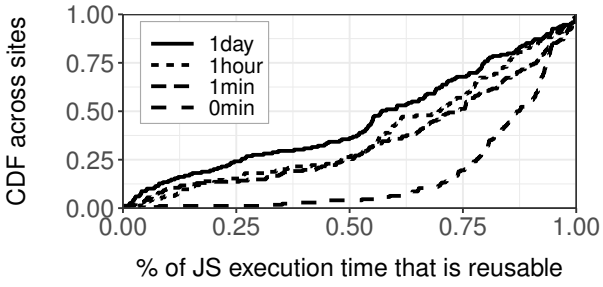
In contrast, if the browser locally caches computations from one of its page loads, the time gap until a subsequent load in which the cached computation is reused can be unbounded. As this time gap grows, operating the cache at the granularity of an entire page becomes increasingly suboptimal since even a small change to page content will render the page-level cached object unusable in subsequent page loads.

To better understand how often and in what ways web pages change over time, we load the landing pages of the Alexa top 500 sites twice, with a 1 hour time gap, and compare 1) the JavaScript source code fetched during each page load, and 2) the final `window` object that represents the constructed JavaScript heap. For source code, we compute the fraction of bytes that are identical, and for the heap, we compute the fraction of keys within the window object, whose values have the same *SHA-1 hash* in both loads. Figure 4 shows that both parts of web computation change over the course of an hour on most pages. A common reason behind this frequent change in page content is the increasing dynamism in modern web pages [31]; web servers often compute responses on-the-fly in order to deliver customized content catered to individual users.

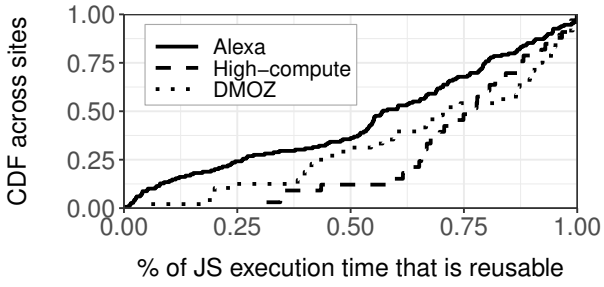
3.2 Our Proposal: Function-Level Caching

While the above results highlight that a page-level caching approach (i.e., a client entirely reuses computation results from a prior page load) will present minimal opportunities for computation reuse,[2] they also show that large parts of JavaScript content remain unchanged across loads of a page. For the median page, Figure 4 shows that 86% of heap state and 89% of JavaScript source code match between two loads separated by an hour; also, JavaScript state

[2]One approach to handling small changes in page content is to patch page-level caching data. However, a client-side caching solution precludes this approach because clients are unaware of content changes until they fully load the latest version of a page (thereby foregoing caching benefits).



(a) Landing pages of 150 sites out of Alexa top 500



(b) Pages loaded at a time gap of 1 day

Figure 5: Reusable JavaScript execution across different time intervals and across different sets of sites.

changes by over 75% across an hour on only 25% of pages. Taken together, there exists significant potential for computation caching, but a fine-grained strategy is necessary to realize the savings.

Determining how fine a granularity to cache at (e.g., small or large code blocks) involves a tradeoff between potential benefits and storage overheads. Finer-grained caching would result in more cache entries and subsequently higher storage overhead, but would also offer more potential benefits (and be less susceptible to page changes).

We observe that a natural solution for balancing this tradeoff is to leverage the fact that most of the JavaScript code on a page is typically within JavaScript functions. On the median landing page among the Alexa top 500 sites, 96% of all the JavaScript code is inside functions. Furthermore, JavaScript functions represent a logical unit of compute as intended by the code developer. These properties naturally lend themselves to a function-level compute caching approach.

4 BENEFITS OF CLIENT-SIDE COMPUTE CACHE

Given the single-threaded nature of JavaScript execution (§2), if a JavaScript function is deterministic, then its execution is reusable when it is invoked with the exact same input state as one of its prior invocations, i.e., outputs from the prior invocation can be applied without executing the function again. In this section, we estimate the potential benefits of client-side computation caching by determining the percentage of JavaScript execution time that can be eliminated by reusing the results of function invocations from prior page loads.

4.1 Overview of JavaScript Function State

A JavaScript function has access to a variety of web page state – global objects, local variables, function arguments, and closures – with the precise set being determined by web security policies (e.g., same-origin policy) and scope restrictions implicit to each state's definition. All of this state is mapped to objects on the JavaScript heap, DOM tree, and disk storage (like localStorage and sessionStorage). Given this, a JavaScript function execution can be summarized by the combination of its 1) *input state*, or the subset of the page's state that it consumes, and 2) *externally visible effects*, such as its impact on the page's global JavaScript heap, calls to internal browser APIs (e.g., DOM), and network fetches.

4.2 Quantifying Potential for Computation Reuse

Methodology: We use the following approach to estimate the benefits of reusing computation from one page load in a subsequent page load. First, we identify all functions which make use of non-deterministic APIs (e.g., Math.random, Date, key traversal of dictionaries, and timing APIs [29]) and network APIs (e.g., XHR requests), and mark all such functions as uncacheable. For all remaining JavaScript functions, during both loads, we track the input state consumed by every invocation. Note that we do not include the input state of the nested functions in the parent functions, instead they are treated as separate function invocations. For each function, we then perform an offline analysis to determine which of its invocations in the later page load had the same input state as an invocation in the initial load; all matching invocations could be skipped via client-side computation caching. We then correlate this information with function execution times reported by the browser's profiler to compute the corresponding savings in raw computation time.

To employ the above methodology, we record web pages with Mahimahi [35] and then rewrite those pages using static analysis techniques [32]. The rewritten pages contain instrumentation code required to track and log function input state; it suffices to log only input states as our goal here is to only estimate the *potential* for reuse, and not to actually reuse prior computations. We then reload each instrumented page in Mahimahi and extract the generated logs. We run our experiments on three different corpora: landing pages for the Alexa top 500 sites, landing pages from 100 (less popular) sites in the 0.5 million-site DMOZ directory [4], and the 86 high-compute sites from Figure 2.

Reuse across loads of the same page. As discussed in Section 3, the time between the page load where the cache is populated and the load where the cache is used to skip function invocations is unbounded. We therefore compare input states for each function invocation across pages loads spaced apart by 1 minute, 1 hour, and 1 day.

Our dynamic tracing tool was successfully able to track input state of JavaScript invocations for 150 of the 500 Alexa pages and 33 of the 86 high-compute pages. Since the distribution of load times across these subsets of sites matches the overall distribution in the respective corpus, we expect our findings to be representative of other pages too. Figure 5(a) shows that, for the median Alexa page, 73% and 57% of JavaScript execution time can be skipped via computation caching across 1 minute and 1 day. From Figure 5(b), the fraction of JavaScript execution that is reusable a day later is higher (76%) for pages which are more in need of computation caching:

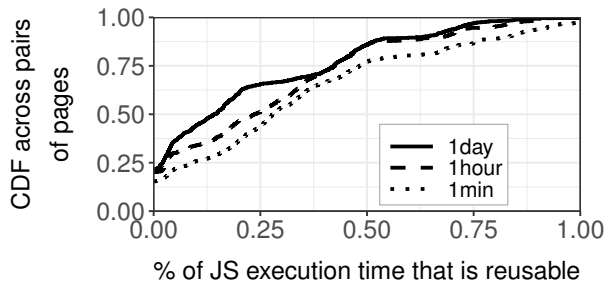


Figure 6: Reusable JavaScript execution across pairs of pages.

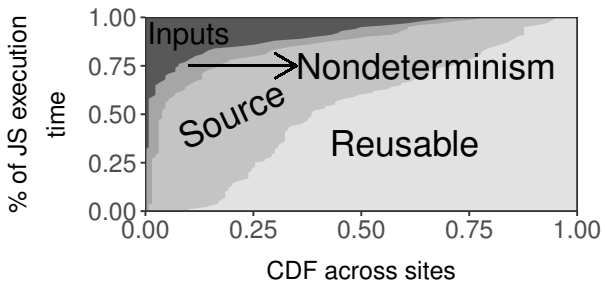


Figure 7: JavaScript execution time breakdown for landing pages of 150 out of Alexa top 500 sites loaded at a time gap of 1 day. Reusable accounts for the execution time that can be reused. Source, Inputs, and Nondeterminism account for execution time which is non-reusable due to change in JavaScript source files, change in function inputs, and non-deterministic functions, respectively.

the high-compute pages.[3] Since JavaScript execution accounts for 65% of client-side computation on the median high-compute page (Figure 2), we estimate a net savings of 49% for the median page. This translates to 990ms of eliminated execution time even on the state-of-the-art Pixel 2 phone. We observe similar reuse potential for the DMOZ pages (48 out of the 100), with a median of 70% of Javascript execution time being reusable across a day.

Reuse across loads of different pages. Thus far, we have focused on reusing computations from a prior load of the exact same page. However, we observe that traditional browser caches support object reuse even across pages. More specifically, object caches are keyed by an object's resource URI, which may appear on multiple pages; this is a common occurrence for pages on the same site, e.g., a shared jQuery library. Inspired by this, we extend our analysis of computation caching to examine how cache entries can be reused across loads of *different* pages from the same site.

We sample 10 sites at random from the Alexa top 1000, and then select 20 random pages on each site. For example, *www.gamespot.com/3-2-1-rattle-battle/* and *www.gamespot.com/101-dinopets-3d/* constitute two pages on the same site. We then compare all pairs of pages on the same site to determine what percentage of JavaScript execution time can be reused between each pair. In other words, we load each of the 20 pages in a site once, load all of these pages again after a time gap, and evaluate how much of the JavaScript execution time on the latter load of a page can reuse executions from prior loads of the other 19 pages. Our tool successfully tracked input states for 80 out of the 200 pages. Figure 6 shows, that for the median pair, 29% and 15% of JavaScript execution time can be reused across time intervals of 1 minute and 1 day, respectively. The % of reuse varies from site to site, with it being as high as 80% for *www.ci123.com* and as low as 9% for *www.prezi.com*.

Note that a user can benefit from all of the reductions in client-side computation that we estimate in this section, both across loads of a page and across loads of multiple pages on the same site, with only modifications to the user's web browser. In contrast, a large number of domains need to adopt prior server-side computation eliding strategies [33] in order for users to see benefits across all of the pages they visit. Furthermore, since JavaScript execution can be reused across pages as well, even the very first load of any page at a client can be sped up by reusing computation from that client's prior loads of other pages on the same site.

[3]To minimize the instrumentation overhead of our tracing tool, we exclude tracking of closure state for high-compute pages. On the subset of these pages which we are able to successfully load with closure state tracking enabled, we see that 59% of JavaScript execution time can be reused a day later on the median page.

4.3 Characterizing Computation Cache Misses

While our results above highlight the significant potential for computation reuse, we see that not all function invocations can be serviced by the cache. To investigate the reason for such misses, we analyze the data for the *"1 day"* line from Figure 5(a), and plot in Figure 7 the breakdown of the total execution time into reusable and non-reusable components (marking the reason precluding reuse).

Non-determinism. Functions which invoke non-deterministic APIs are not amenable to computation caching. Figure 7's *"Nondeterminism"* shaded area shows the amount of execution time marked non-reusable due to the presence of non-deterministic functions on the page. For the median page in our corpus, 0.5% of total non-reusable time was accounted for by such non-deterministic functions.

Changes to JavaScript source code. For a function's execution to be reusable from an earlier page load, the source code of the JavaScript object file containing the function's definition should not have changed since that load.[4] As previously shown in Figure 4, 11% of JavaScript source code is subject to change within a time period of one hour. The longer the time gap between loads, the more susceptible are web pages to load object files with different source code [46].

Figure 7 shows that change in source code is the predominant reason for non-reusable computation. Further analysis reveals that for 13% of pages, change in source code was the only reason hindering reuse.

Changes to input state. Figure 7 also shows the percentage of non-reusable computation accounted for by changes in the input state of a function. A JavaScript function can observe different sets of inputs across invocations on different page loads, either due to changes in server-side state or due to non-determinism on the page.

Changes in server-side state can influence the responses sent to the client, which in turn can affect the input state of JavaScript invocations. For example, *www.cnblogs.com* sends a cookie known as *RNLBSERVERID* as a part of the response to a client request. This cookie value is used for server-side load balancing. Since the value of this cookie changes dynamically depending on server logic, a

[4] Though it would suffice for only the source code within the function's boundaries to remain unchanged, our analysis considers cache entries for a function as non-reusable if the file containing the function changes.

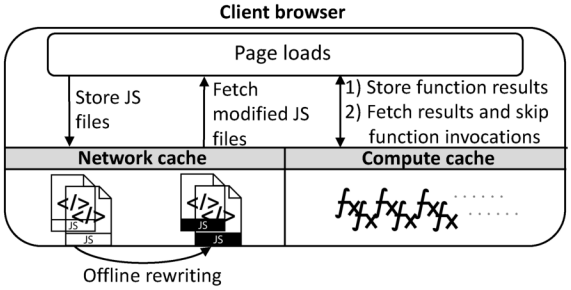


Figure 8: High level proposed design for enabling client-side reuse of page load computations.

JavaScript function on this page reads different values for this cookie across loads spread out by 1 hour or more, causing this function to forego the use of results from prior invocations.

Inputs to a function can also change across loads due to non-determinism in functions which are invoked earlier in the page load. In order to distinguish between server-side state and non-determinism as the cause for change in input state, we load the same recorded set of webpages twice using Mahimahi. This eliminates any potential changes in the source code of JavaScript object files and server-side state. We then compare the input state across these two loads. Any mismatch could only be due to non-determinism on the page. Figure 5(a)'s *"0min"* line shows that for the median page, 18% of the total execution time is non-reusable due to such non-determinism.

5 ENVISIONED SYSTEM AND CHALLENGES

While our primary goal in this paper is to motivate the need for client-side solutions and quantify the potential benefits of our proposed client-side reuse of JavaScript executions, we end by describing how web browsers might implement a computation cache and the challenges in doing so.

5.1 System Workflow

Figure 8 illustrates our envisioned approach. When the user loads a page, the browser would fetch and cache previously un-cached resources, as it does today. The browser can then use (offline) static program analysis to instrument the cached scripts to track the state that each JavaScript function accesses or modifies (Section 4.1). Finally, when an instrumented script is executed in any page load, prior to executing every instrumented function, the browser can perform a computation cache lookup in search of a matching entry (i.e., a prior invocation of the same function with the same inputs); if a match is found, the browser would apply the effects from that cache entry instead of executing the function; otherwise, it would execute the function and add a new entry to the cache.

5.2 Practical Challenges

Although our proposed design requires only browser modifications (easing deployability), our data collection in Section 4.2 reveals several overheads that hinder practicality.

- *Tracking:* The instrumentation required to track per-function state accesses and modifications can result in user-facing slowdowns of up to 200× [19]. To mask these overheads, tracking could be performed offline or using spare CPU cores (§2), but such solutions would require special care (designed per page) to avoid negatively impacting persistent client- or server-side state (e.g., cookies) via, say, idempotent functions [47].
- *Cache management:* Similar to the browser's network cache, the computation cache would incur various caching overheads in terms of lookup times and storage. These overheads are exacerbated in the case of computation cache because of differences in the granularity at which the two caches operate: cache entry per resource in the network cache versus cache entry per function invocation in the compute cache. On the median landing page across the Alexa top 500 sites, the number of function invocations per page load (close to 9K) are more than two orders of magnitude higher than the number of resources fetched (30). Therefore, the overheads associated with the compute cache are likely to be much higher than with the network cache.

Solution. To address these overheads, we make the following key observations: 1) the distribution of execution time across JavaScript functions on a page is (typically) heavily skewed towards only a small fraction of functions, and 2) these small fraction of functions have relatively stable execution times across page loads. For the high-compute pages in our corpus, 80% of execution time on the median page is accounted for by only 27% of the functions. Across loads separated by an hour, the execution time for the median of these functions changed by only 15%. This implies that, to sidestep the aforementioned overheads without sacrificing most reuse benefits, only the subset of functions accounting for the bulk of the execution time should be considered for caching. Note that function-level execution time information can be easily extracted using built-in browser profilers [22] at low overheads.

6 CONCLUSION

Despite an abundance of proposed solutions, client-side computation continues to slow down mobile web page loads. To overcome this, we propose a purely client-side solution to elide web computations: augmenting web browsers with a computation cache. As a first step towards this vision, we empirically motivate the need for a finer granularity of computation reuse than prior systems, and we quantify the potential for client-side reuse of JavaScript function executions. We estimate that 49% of client-side computation can be reused across loads even a day apart on the median high-compute web page. We outlined a potential system architecture to realize our proposed client-side web computation cache and the challenges entailed in reaping the estimated benefits.

The mobile web landscape is highly dynamic, with the underlying technology changing at a rapid rate. In light of this dynamism, it is hard to predict the applicability of a technique, such as ours, even a few years down the line. However, we believe that certain web trends will continue to hold, as they have in the past, and would therefore continue to enable JavaScript computation reuse. First, prior studies [37, 49] have shown that more than half the web pages browsed by users are revisits. We expect page revisits to be as common going forward. Second, despite the web's dynamism, it has been shown that large parts of web page content are fairly static [27, 46]. Taken together, we believe there will continue to exist significant opportunity for reusing JavaScript computations as a means to improve mobile web performance in the long run.

REFERENCES

[1] 2016 Q2 mobile insights report. http://resources.mobify.com/2016-Q2-mobile-insights-benchmark-report.html.

[2] Browser market share worldwide. https://gs.statcounter.com/browser-market-share.

[3] Cisco annual Internet report highlights tool. https://www.cisco.com/c/en/us/solutions/executive-perspectives/annual-internet-report/air-highlights.html.

[4] Directory of the web. https://dmoz-odp.org/.

[5] Find out how you stack up to new industry benchmarks for mobile page speed. https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks/.

[6] HTTP/2 adoption. https://httparchive.org/reports/state-of-the-web#h2.

[7] Network latency numbers. https://github.com/WPO-Foundation/webpagetest/blob/master/www/settings/connectivity.ini.sample.

[8] Smartphones are driving all growth in web traffic. https://www.vox.com/2017/9/11/16273578/smartphones-google-facebook-apps-new-online-traffic.

[9] State of JavaScript. https://httparchive.org/reports/state-of-javascript.

[10] Why performance matters? https://developers.google.com/web/fundamentals/performance/why-performance-matters.

[11] FCC: Broadband performance (OBI technical paper no. 4). https://transition.fcc.gov/national-broadband-plan/broadband-performance-paper.pdf, 2009.

[12] Third annual broadband study shows global broadband quality improves by 24% in one year. https://newsroom.cisco.com/press-release-content?type=webcontent&articleId=5742339, 2010.

[13] GSMA: The State of Mobile Internet Connectivity 2020. https://www.gsma.com/r/wp-content/uploads/2020/09/GSMA-State-of-Mobile-Internet-Connectivity-Report-2020.pdf, 2020.

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

[15] Amazon. Silk Web Browser. https://docs.aws.amazon.com/silk/latest/developerguide/introduction.html, 2018.

[16] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: MapReduce for incremental computations. In *SoCC*, 2011.

[17] K. Boos, D. Chu, and E. Cuervo. Flashback: Immersive virtual reality on mobile devices via rendering memoization. In *MobiSys*, 2016.

[18] M. Butkiewicz, D. Wang, Z. Wu, H. Madhyastha, and V. Sekar. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*, 2015.

[19] A. Chudnov and D. A. Naumann. Inlined Information Flow Monitoring for JavaScript. In *CCS*, 2015.

[20] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile Internet QoE. In *IMC*, 2018.

[21] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet QoE. In *IMC*, 2018.

[22] G. Developers. Chrome DevTools. https://developers.google.com/web/tools/chrome-devtools/.

[23] D. F. Galletta, R. Henry, S. McCoy, and P. Polak. Web site delays: How tolerant are users? *Journal of the Association for Information Systems*, 2004.

[24] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, 2010.

[25] L. Huston, R. Sukthankar, R. Wickremesinghe, M. Satyanarayanan, G. R. Ganger, E. Riedel, and A. Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST*, 2004.

[26] W. Lee, E. Slaughter, M. Bauer, S. Treichler, T. Warszawski, M. Garland, and A. Aiken. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC*, 2018.

[27] S. Mardani, M. Singh, and R. Netravali. Fawkes: Faster mobile page loads via app-inspired static templating. In *NSDI*, 2020.

[28] D. Michie. "Memo" functions and machine learning. *Nature*, 1968.

[29] J. Mickens, J. Elson, and J. Howell. Mugshot: Deterministic capture and replay for javascript applications. In *NSDI*, 2010.

[30] J. Nejati and A. Balasubramanian. An in-depth study of mobile browser performance. In *WWW*, 2016.

[31] J. Nejati, M. Luo, N. Nikiforakis, and A. Balasubramanian. Need for mobile speed: A historical analysis of mobile web performance. In *TMA*, 2020.

[32] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *NSDI*, 2016.

[33] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *NSDI*, 2018.

[34] R. Netravali and J. Mickens. Remote-Control Caching: Proxy-Based URL Rewriting to Decrease Mobile Browsing Bandwidth. In *HotMobile*, 2018.

[35] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, J. Mickens, and H. Balakrishnan. Mahimahi: Accurate Record-and-Replay for HTTP. In *USENIX ATC*, 2015.

[36] P. Norvig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[37] H. Obendorf, H. Weinreich, E. Herder, and M. Mayer. Web page revisitation revisited: implications of a long-term click-stream study of browser usage. In *CHI*, 2007.

[38] Opera. Opera Mini. http://www.opera.com/mobile/mini, 2018.

[39] G. Phillips. Smartphones vs. desktops: Why is my phone slower than my PC? https://www.makeuseof.com/tag/smartphone-desktop-processor-differences/.

[40] C. Radoi, S. Herhut, J. Sreeram, and D. Dig. Are Web Applications Ready for Parallelism? In *PPoPP*, 2015.

[41] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *SIGCOMM*, 2017.

[42] A. Sivakumar, C. Jiang, S. Nam, P. Shankaranarayanan, V. Gopalakrishnan, S. Rao, S. Sen, M. Thottethodi, and T. Vijaykumar. Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Mobicom*, 2017.

[43] A. Sivakumar, S. Puzhavakath Narayanan, V. Gopalakrishnan, S. Lee, S. Rao, and S. Sen. PARCEL: Proxy assisted browsing in cellular networks for energy and latency reduction. In *CoNEXT*, 2014.

[44] A. Suresh, E. Rohou, and A. Seznec. Compile-time function memoization. In *CC*, 2017.

[45] A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec. Intercepting functions for memoization: A case study using transcendental functions. *ACM Transactions on Architecture and Code Optimization (TACO)*.

[46] X. S. Wang, A. Krishnamurthy, and D. Wetherall. How much can we micro-cache web pages? In *IMC*, 2014.

[47] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding Up Web Page Loads with Shandian. In *NSDI*, 2016.

[48] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why are web browsers slow on smartphones? In *HotMobile*, 2011.

[49] H. Weinreich, H. Obendorf, E. Herder, and M. Mayer. Not quite the average: An empirical study of web use. *ACM Transactions on the Web*, 2008.