NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks

Ashiwan Sivakumar
Purdue University
West Lafayette, IN, USA

Chuan Jiang
Purdue University
West Lafayette, IN, USA

Yun Seong Nam
Purdue University
West Lafayette, IN, USA

Shankaranarayanan P.N.
AT&T Labs – Research
Bedminster, NJ, USA

Vijay Gopalakrishnan
AT&T Labs – Research
Bedminster, NJ, USA

Sanjay G Rao
Purdue University
West Lafayette, IN, USA

Subhabrata Sen
AT&T Labs – Research
Bedminster, NJ, USA

Mithuna Thottethodi
Purdue University
West Lafayette, IN, USA

T.N. Vijaykumar
Purdue University
West Lafayette, IN, USA

ABSTRACT

Despite much recent progress, Web page latencies over cellular networks remain much higher than those over wired networks. Proxies that execute Web page JavaScript (JS) and push objects needed by the client can reduce latency. However, a key concern is the scalability of the proxy which must execute JS for many concurrent users. In this paper, we propose to scale the proxies, focusing on a design where the proxy's execution is solely to push the needed objects and the client completely executes the page as normal. Such *redundant execution* is a simple, yet effective approach to cutting network latencies, which dominate page load delays in cellular settings. We develop *whittling*, a technique to identify and execute in the proxy only the JS code necessary to identify and push the objects required for the client page load, while skipping other code. Whittling is closely related to program slicing, but with the important distinction that it is acceptable to approximate the program slice in the proxy given the client's complete execution. Experiments with top Alexa Web pages show *NutShell* can sustain, on average, 27% more user requests per second than a proxy performing fully redundant execution, while preserving, and sometimes enhancing, the latency benefits.

CCS CONCEPTS

• **Networks** → **Transport protocols**; *Network measurement*; *Cloud computing*; • **Information systems** → *Browsers*;

KEYWORDS

Mobile Web, Cloud computing, Proxy-assisted browsing, Program Slicing, Whittling

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MobiCom '17, October 16–20, 2017, Snowbird, UT, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4916-1/17/10…$15.00
https://doi.org/10.1145/3117811.3117827

1 INTRODUCTION

Web pages have, over the years, evolved from simple and relatively static pages to ones that are feature rich and customized to individual user preferences. This evolution, however, has made them significantly more complex [17, 33], with most pages comprising of tens to hundreds of static and dynamic objects (images, cascading style-sheets (CSS), JavaScript (JS) files, etc.) downloaded from multiple domains. Consequently, today's Web page download process involves many HTTP request-response interactions, each triggered due to the parsing of, or interpretation of one or more objects on the page. When network latencies go up, as is typically the case with cellular networks, the page load times increase significantly degrading user experience. Users, on the other hand, have come to expect an interactive experience to the point where studies show revenue losses due to poor responsiveness [1, 3, 4, 24].

Many recent attempts [27, 44], most notably SPDY [27] which has shaped the recent HTTP/2 standard [32], has attempted to address protocol level limitations with traditional HTTP. HTTP/2 and SPDY seek to accelerate page loads by allowing for multiple outstanding requests in parallel on a single connection, and supporting out-of-order delivery of responses. However, the performance improvements of these protocols in the real world are mixed [25, 55]. A key reason is that the objects needed for the page to load cannot be requested in parallel because of complex dependencies in pages (Figure 1b). To overcome this limitation, the protocol allows the server to push objects to the client without waiting for explicit client requests. However, server push requires explicit identification of objects that can be pushed. This constraint is non-trivial since many Web pages require parsing/executing HTML, CSS and JS to identify the associated objects.

Recent proposals suggest the use of a powerful, well-connected proxy that can emulate (part of the) client functionality, including JS execution, and push the required objects to the client. The proxy's functionality (and hence complexity) can vary depending on the solution. In one approach [42, 52], the client performs all of the functions of a traditional client, while proxies perform *redundant execution*, merely to identify and push objects needed by the client. In another approach, the proxy generates a "rendered" page that the client can display with minimal work [6, 11, 56, 62]. Here, proxy computation is not redundant, and client-side processing is

completely [6, 11, 62], or partially [56] eliminated. Results show that both redundant and non-redundant execution can significantly reduce page load times. While non-redundant execution promises additional benefits by reducing or eliminating client-side compute, it may result in additional latencies on client interactions [51] for the complete elimination approaches, or additional complexity associated with migrating execution state mid-flight from the proxy to the client for the partial elimination approaches.

Regardless of the proxy design choice, their benefits are at the cost of significant additional computational overheads, a dominant component of which is JS execution. Deploying such proxies at carrier-scale to millions of users requires that the computational overheads of the approach be economized (our measurements indicate a server with 32 cores and 128 GB RAM can support 2000 users, which translates to $2.5 Million in CAPEX alone for a million users assuming $5000 per server). Motivated by this scaling challenge, in this paper we focus on minimizing the proxy computational overheads in general, and JS execution in particular. We tackle this challenge in the context of redundant execution, since it is simple, allows for responsive client interactions and is well-suited for cellular settings where the network constitutes more than half the client latency (§2).

We present the design of *NutShell*, a system to tackle these challenges. *NutShell* leverages two key observations. First, the proxy need not execute all JS code (e.g., UI-related code need not be executed). Instead, only the subset of code necessary to identify and fetch the objects to be pushed is executed (see Fig. 2 for a detailed example). In other words, using terminology used in the programming language community, only the *backward slice* of the code [53, 58] related to URL fetching must be executed. Second, while static analysis of JS code is a hard, open research problem (e.g., [26, 34, 48, 54]), proxies can approximate the backward slice since only the redundant proxy execution is affected. Because the client performs the actual full execution, the client would directly fetch any objects not pushed by the proxy, trading off client latency for computation at the proxy without any correctness problems.

Since statement-level slicing is time-consuming, our approach, called *whittling*, works at JS function granularity, turning off entire functions that do not affect the set of fetched objects. Thus, whittling *dynamically learns* the slices by turning off function definitions – *i.e.* all the invocations of a function, whereas conventional slicing selectively turns off individual invocations. The approach is complicated by two issues. First, examining all JS functions would be time-consuming. Second, owing to inter-function dependencies, turning off two functions simultaneously may impact object fetching, although turning them off individually may not. We tackle the first issue by exploiting the fact that a majority of JS execution time is spent in a small fraction of heavy functions. We identify the heavy functions via profiling and examine only those functions. Because identifying the optimal set of independent functions would be time-consuming, we tackle the second issue with a greedy approach of examining functions in the decreasing order of their execution times, to grow a set of functions that may be turned off together. Finally, despite the above optimizations, whittling at every page load would be too slow to be effective. To that end, we exploit the fact that although objects in a page change, the code is stable over a period of several hours to allow profitable reuse of the same slice over several loads of a given page.

Our contributions are:

- presenting the first effort to our knowledge for scaling execution-based Web proxy designs.
- proposing a dynamic learning scheme, called *whittling*, to compute approximate backward slices of object fetches at function granularity; and
- proposing several optimizations to make whittling computationally efficient, practical and effective.

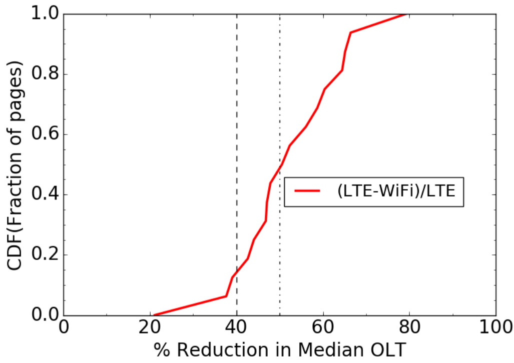
We conducted experiments with 78 pages from the Alexa Top 100 Web-sites. Our key results are:

- *NutShell* reduces JS computation by 1.33X in the median case, and up to 4X for some pages. Further, the user requests per second increases on average by 27% for a range of web page popularity models, and upto 4X for some pages.
- The scalability benefits can be achieved while preserving, and even exceeding the latency gains of a redundant execution approach. By combining redundant execution and whittling, *NutShell* achieves speedups in median page load times of 1.5 compared to SPDY and speedups of 20% compared to fully redundant execution for 15% of the pages.
- Whittling can be computed in an online fashion. Through a longitudinal study we show that for 92% of the Web pages *NutShell*'s whittling remain accurate (i.e., it pushes all the needed objects) over 3-hour windows. Further, the whittled JS can be reused across users due to large code overlap.

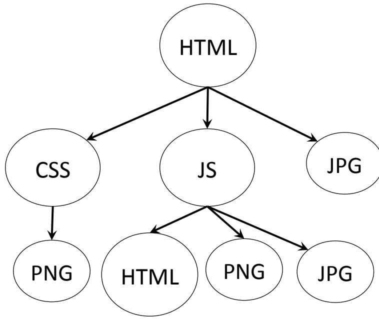
2 MOTIVATION

The overall client latency comprises compute delay (for parsing and executing HTML, CSS and JS, and for rendering) and network delay to fetch the required objects. For mobile devices over cellular networks, which is our focus, the network component is a dominant component of the overall latency. To see this, consider Fig. 1a which shows the reduction in the Onload time for a mobile phone when moving from a cellular LTE connection to a Wi-Fi connection for 20 top Alexa pages. The Onload time (OLT) is a common measure of page load latency, and is the time from request initiation until when the browser triggers an onLoad() event. Fig. 1a shows that the OLT reduces by more than 50% for 53% of the pages, and by more than 40% for 82% of the pages. Since the compute activity in both cases is the same, these percentages directly relate to the network component of the overall client latency. The network component may in fact be higher because a portion of the latency with WiFi could also be attributed to network activity.

A commonly used technique to reduce the network component is push (which controls what objects are pushed to the client without explicit requests). A key difficulty with any push technique is that owing to dependencies inherent in Web pages (Figure 1b), objects required later in the page load process can be identified only after the execution of prior objects (e.g., JS). A common approach then is to push objects whose URLs are embedded in the root HTML (often referred to as *embedding level 1* (or simply L1) objects [55]). However, the approach is limited by the fact that L1 objects only constitute a subset of all objects. Further, since the HTML may include objects from multiple domains, in practice not



(a) % reduction in OLT for mobile page loads when moving from LTE to Wi-Fi.



(b) Dependency graph of a page load – JS execution may trigger further object fetches.

Figure 1: Motivation

all L1 objects can be pushed. In an *execution-based* approach, a proxy [5, 6, 11, 42, 52, 56, 62] identifies all objects that the client needs by parsing HTML and CSS, and executing JS. The proxy, with much faster network connectivity, and secondarily faster compute, can quickly identify and fetch the objects needed for a client's page load, and proactively push *all* the objects to the client, so the network delay associated with explicit client requests can be avoided (§6.3 experimentally shows the latency benefits).

An alternative to a proxy-based execution approach is to observe which objects are fetched across users of a page load, and push those objects. Given that pages are often personalized, such an approach can only push content common across users. To evaluate the potential of this approach, we conducted a user study with 8 real users simultaneously downloading a series of landing pages from Alexa top 100 (refer §6.4 for further details) and assume only common objects fetched across downloads of the same page by the 8 different users are pushed, with all other objects explicitly pulled by each user. For 29% of the pages, the median % of objects that must be pulled across user loads exceeds 55%, while for 50% of the pages, the median % of objects that must be pulled across user loads is 31%. This indicates that relying on historical observations of objects across page loads can miss out on latency savings offered by proxy-based execution since only a subset of objects can be pushed by the proxy. That said, it may be possible to combine such an approach with proxy-based execution as we discuss in §8.

Execution based approaches themselves differ based on whether they eliminate client JS execution. Eliminating client JS execution

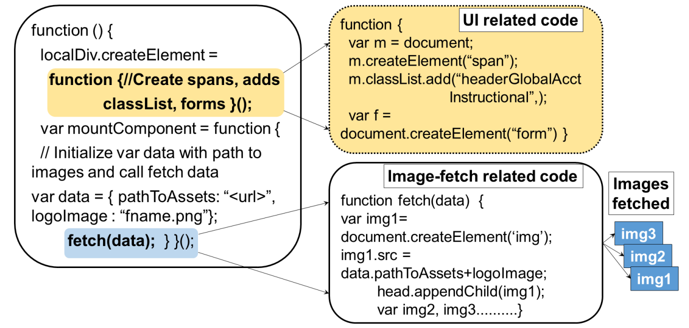


Figure 2: Whittling Example

has the advantage of reducing client computation related delays, but has associated trade-offs that we detail in §7. Regardless of these differences, a common unaddressed challenge to all these designs is the computation scaling bottlenecks associated with execution based approaches. In this paper, we focus on tackling the proxy computation bottlenecks in the context of *redundant execution* approaches [42, 52] which do not eliminate any client-side execution. Instead, the proxy executes redundantly only for identifying objects needed by the client. As such, the client execution remains unchanged except for seeing faster object fetches. We focus on redundant execution given its simplicity and effectiveness in reducing network delay which dominates cellular client latencies.

Finally, a potential approach to reducing the computation requirements at the proxy is to only perform redundant execution for a subset of the most popular pages. However, such an approach can lose out on the latency benefits of redundant execution for a large number of pages (as we show in §6.1). In contrast, our goal in this paper is to reduce the proxy computation requirements, while still preserving the latency benefits of redundant execution for a larger set of pages.

3 NUTSHELL DESIGN

NutShell seeks to scale proxies based on redundant execution by addressing their primary computation bottlenecks. Given that browser functionality such as rendering and display are not replicated at the proxy, and parsing HTML and CSS is relatively light-weight, the dominant portion of proxy computation is the execution of JS [12]. *NutShell* leverages the key insight that since proxy execution is redundant, it suffices for the proxy to execute only the JS code necessary to fetch objects. *NutShell*'s central mechanism – whittling – effectively removes JS code that does not affect the URLs fetched.

To illustrate the opportunity for whittling, Fig. 2 shows a concrete example of the JS from an Alexa Top 100 Web page. A top-level function (left side of Fig. 2) calls two functions (right half of Fig. 2). One of the functions sets up the UI-related aspects of the page such as span creation, button creation and addition of event listeners to handle button clicks. This function does not affect the fetching of objects. The second function fetches a number of images which are displayed in the UI panels. The UI-related function (shaded rectangle with dotted outline in the top-right corner of Fig. 2) can be whittled at the proxy without affecting the set of fetched objects. The top-level function and the image fetching functions (clear boxes

with solid outlines in Fig. 2) are in the backward slice of the fetched objects, and hence must be preserved.

NutShell's use of whittling is related to the area of program slicing, which has seen much research in the programming languages and compiler community in the last couple of decades [53, 58]. Despite promising advances (e.g., [26, 34, 48, 54]), computing program slices for JS code using *static analysis* techniques remains a hard problem in general. Specifically, JS's use of dynamic typing and `eval` could result in backward slices whose sizes approach the size of the original program, diminishing their effectiveness in reducing computation overheads – one of our key goals.

Instead, *NutShell* employs dynamic learning of backward slices, which involves comparing URLs fetched before and after statements of code are dropped. Although such an approach has been explored in other contexts (e.g., finding language-independent program slices and for fault isolation) [15, 23, 61], *NutShell*'s context presents unique opportunities. Unlike fault isolation where false negatives (i.e., missing a fault) are unacceptable, in *NutShell*'s context it is acceptable to approximate the backward slice. Such acceptability arises from slicing being performed only on the redundant *NutShell* proxy execution. Since the client performs the actual full execution, any objects not pushed by the *NutShell* proxy could be fetched directly by the client, trading off client latency reduction opportunities for computation savings at the proxy.

We next describe *NutShell*'s whittling strategy which uses a dynamic approach, but with the ability to tolerate imperfect (approximate) slicing.

3.1 Whittling individual functions

NutShell makes the design choice to whittle code at the function granularity. The choice of function granularity is driven by the opportunity-overhead trade-off; choosing fine-granularity (e.g., statements) may provide the ability to whittle additional code, but may increase the overhead because each statement may have to be individually tested. On the other hand, coarse-grained whittling of entire JS files results in minimal benefits as most files cannot be whittled if even a single function affects object loading. Further, the relatively-few JS files that can be whittled are typically rarely-executed files that do not result in significant savings even if whittled. Note that whittling eliminates all dynamic invocations of the function. This implies that we are conservative in function whittling; if even one invocation of the function affects URL fetches, the function will not be whittled.

NutShell uses automated two-version testing to determine if a function can be safely whittled while ensuring that all objects needed for page load are fetched. Our mechanism generates two versions of a page: (i) an unmodified *full* version 'F'; and (ii) a *partial* version 'P' produced by eliding the function under test by rewriting the function to be an empty function. If both versions identify the same set of objects for downloading, then the function under test can be whittled because eliding it does not affect the objects fetched.

Determining the set of objects fetched by the baseline 'F' version itself poses interesting issues, since multiple interpretations are possible regarding when a page load is considered complete. Nominally, one can consider a page load to be complete based on time bounds (e.g., after 30 seconds). Alternately, recognizing that objects needed for an initial acceptable rendering of a page are more critical to user experience than other objects, a page may be viewed complete based on browser events (e.g., when the browser onLoad event fires), when all above-the-fold content is loaded [7] or when content with the highest utility to users is received[18, 35].

NutShell is agnostic about the metric of page load completion; however, for any chosen metric, an appropriate *signature*, must be extracted, which is the subset of objects fetched by the 'F' version that serve as a baseline of comparison for the whittling tests. To be concrete, we use the browser onLoad event to determine page completion. We run the 'F' version many times till onLoad and use the intersection set of objects fetched in each run as the *signature* of the page. Doing so ensures the signature only contains objects always fetched before onLoad (note that in any given run, additional URLs may be fetched incidentally as a consequence of asynchronous JS). *NutShell* may be extended in the future to accommodate other notions of page completeness.

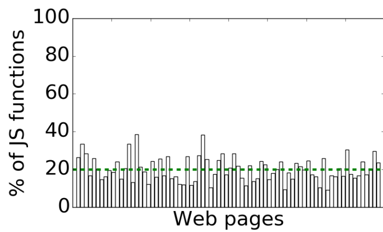
Our two-version test outcome determines whether the function may be safely whittled (i.e., the signature matches) or not (i.e., there is a mismatch). Note that even in cases where there is a match, there may be other side-effects due to function whittling. For example, whittling a function may give rise to errors because some objects (which would be defined in the whittled function) are undefined. Such errors impose some minimal overheads as the errors must be caught/handled (often with a nominal error message output to the console). Such errors do not affect our technique as (1) our focus is solely on whether all the objects in the signature are fetched, and (2) we fully include the overheads of error handling in our measurements.

All of *NutShell*'s two-version testing is performed in a recorded environment. The first access to the page by the proxy (where all JS is executed) is recorded, and all testing of JS subsets occurs by replaying the recorded page in a deterministic manner [8, 42, 52]. Doing so ensures the whittling tests are not impacted by randomization, and date/time-dependent code which may complicate ascertaining whether the differences between the F and P versions are because of whittling or because of variability.

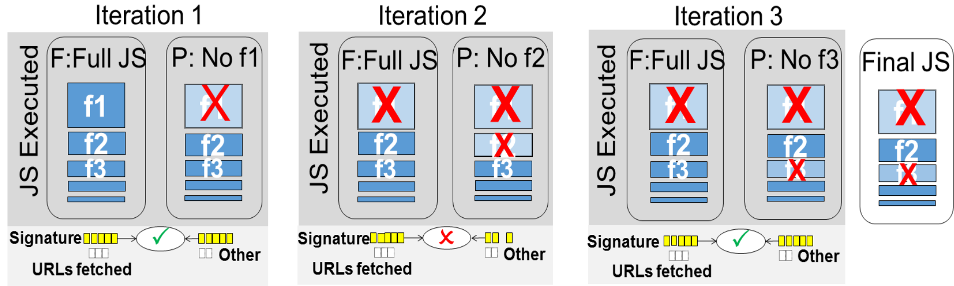
3.2 Whittling across functions

With the above mechanism, we can test any individual function to determine if it may be whittled. However, directly using the approach to test *all* functions has two weaknesses. First, Web pages often have hundreds of JS functions, many of which are rarely invoked. Testing them all increases overheads without commensurate benefits. Second, there are often dependencies among JS functions that prevent collective whittling of multiple functions even though each function may be whittled individually.

To avoid testing all functions, *NutShell* employs a greedy heuristic by sorting functions in the order of their computational work (captured by execution time). We measure the work done in each function by profiling a full JS execution. By testing functions in the order of computational effort, we maximize the potential savings from whittling. The greedy order is especially effective because we observed that on an average 20% of JS functions account for 80% of JS execution time across all Alexa top 100 pages. Fig. 3a shows the percentage of functions that account for 80% of JS CPU-time for the



(a) Percentage of functions accounting for 80% of total JS execution time.



(b) Walk-through example of NutShell's Greedy Whittling.

Figure 3: NutShell's Whittling Method

Alexa Top 100 pages(X-axis). Uniformly, we observe that a small percentage (9%-38%) is enough to cover 80% of execution time. This 80-20 rule enables us to limit whittling to this percentage.

Handling Dependencies: Consider the following example taken from an Alexa Top 100 Web page. The JS code contains two functions (say A and B) both of which invoke the jQuery initializer. The jQuery initializer invocation impacts other URL-fetching code and hence is needed. The other work in functions A and B are not relevant for any URL fetch. When doing the basic whittling test, we find that each function is individually safe to whittle because jQuery initializer is still invoked in the other function. However, when both functions are whittled, the jQuery initializer is never invoked, which affects other parts of the JS code which fetch URLs.

The above example is one of many possible dependencies that prevent whittling of large collections of functions. Because such dependencies are hard to analyze, we take an empirical approach. Specifically, we use the greedy order of function testing to grow a set of functions that may be turned off together. We describe this greedy algorithm using an example (Fig. 3b).

Fig. 3b assumes a Web page with the JS functions pre-sorted in decreasing order of computation effort (f1, f2, f3, and so on). We start with a two-version test against the full Web page load which includes execution of all JS ('F' version in iteration 1 of Fig. 3b). If the 'P' version which whittles f1 results in the same signature as that of the 'F' version, the function f1 is whittled/dropped from future runs. Subsequent functions are further tested to see if they can be whittled in conjunction with all the previously-whittled functions. These secondary tests are an alternative two-version test in which the F version is the JS code *without* all previous functions that can be safely whittled (as determined by previous tests) and the P version which drops the new function that is under test. (For example, in iteration 2, the 'F' version whittles f1 because f1 is known to be safe to drop from the previous iteration. The 'P' version additionally drops f2 to test if f2 may be safely whittled.) Functions that can be dropped without impacting the page signature are marked for whittling; others must remain in the executed JS. In Fig. 3b, f2 cannot be whittled (because of signature mismatch when f2 is dropped), but f3 can be whittled. This process continues to whittle the JS code until all functions under consideration for whittling are tested. *NutShell*'s greedy approach has the added advantage that it minimizes the overheads of dynamic learning as the heavier functions are whittled for a large fraction of tests. As a practical matter, implementations may choose to filter the set of functions that are considered based on (a) minimum work threshold, to avoid examining light functions that do not provide significant benefits, and (b) numerical limits, to bound the time overheads of whittling. (In practice, *NutShell* tests up to 200 of the top functions till we account for 80% of CPU work.)

NutShell's greedy heuristic strategy does not allow for backtracking (e.g., by bringing back a dropped function); the set of dropped functions starts with the heaviest function that can be whittled and can only grow by adding other functions that can be whittled without dependency problems. As such, the result may not be optimal. However, we show later that the greedy heuristic is effective in practice. More sophisticated techniques to identify collections of functions that may be simultaneously whittled, which we leave for future work, can only improve *NutShell*'s results.

3.3 Amortizing overheads across page loads

The process of dynamically learning the whittled JS depends on (1) the number of functions, which ranges from the low 10s to 200, and (2) the time for each per-function two-version test, which is typically a few seconds because each test is a page load (0.3-4s) followed by signature comparison (<10 ms). For our evaluation set of Web pages from Alexa Top 100 (§5), the average learning time is 213 seconds across pages.

The dynamic learning of JS whittling is not done for every page load. Rather, *NutShell* performs whittling for the first page load, and then re-uses the whittled code for all the common JS content in a new load. §4 discusses how we implement such reuse. In this section, we present an empirical study showing the feasibility of such reuse. The study is based on a recording of 25 pages from the Alexa Top 100 obtained every hour over a 24 hour period using the approach described in §5.

For each page, we whittled JS based on the version recorded at time $t = 0$. We extract the signature (the set of objects needed for page load) for the $t = 24hr$ recording based on a full execution of all JS in that recording. We then determine the fraction of objects in the signature fetched by the whittled JS code using whittling learnt at $t = 0$. Fig. 4 shows the corresponding fractions. For all but 2 pages, 99% of the signature or higher can be fetched, indicating whittling reuse is effective even over a 24 hour period for most pages.

Note that while the JS is stable the page content is not. Fig. 5 shows the incremental differences in the page signatures at time $t = 0$ and $t = 24hr$ for the pages on the Y-axis. Objects that are

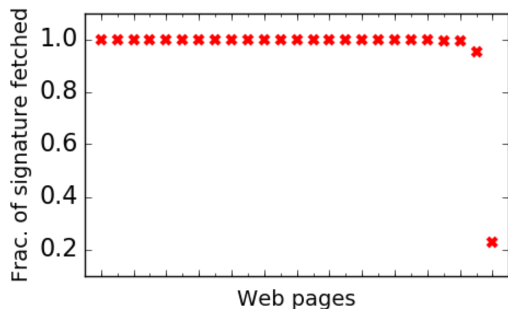


Figure 4: Fraction of signature URLs fetched by whittled JS at time t=24hr using whittling learnt at t=0.

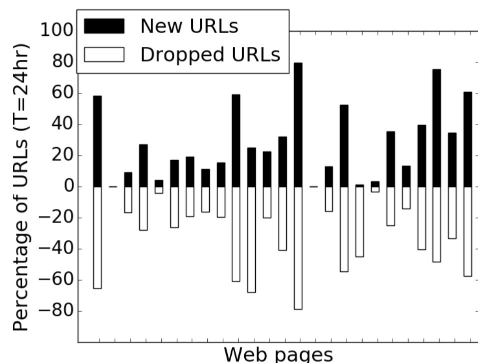


Figure 5: Percentage of new and dropped URLs in the page signature at t=24hr compared to t=0.

fetched at $t = 24hr$ that were not present at $t = 0$ are shown on the positive side. Objects that were fetched at $t = 0$ that were absent at $t = 24hr$ are shown on the negative side. Even though there is significant churn in the Web page content over a period of 24 hours, executing the same 24-hour-old whittled JS is effective at fetching the changed content.

In practice, it is acceptable to relearn whittling over more frequent time intervals. For instance, reusing whittled code over a 3-hour window results in under 2% overhead ($2\% = 213s/(3hr \times 3600secs/hr)$), for a learning time of 213s discussed above. Even for the page which had more frequent changes (rightmost bar in Fig. 4), changing *NutShell*'s learning frequency to once every 3 hours results in a larger fraction of objects being fetched.

For pages with dynamically changing JS, *NutShell* may lose some of the CPU savings from whittling (and the resulting throughput improvement at the proxy) because the functions identified for whittling may not be present in the changed JS. To evaluate this concern, we consider the fraction (%) of functions (X-Axis) that can be whittled based on the $t = 0$ version, which are still relevant for whittling at a later time across the pages. For the $t = 3hr$ page load, for 75% of pages, all functions can be whittled, while for another 15% of pages over 70% of functions can be whittled. Note however that as Fig. 4 shows, client latency is not affected for most pages because the proxy fetches and pushes all objects obtained from its JS execution. Overall, these results show that it is viable to reuse

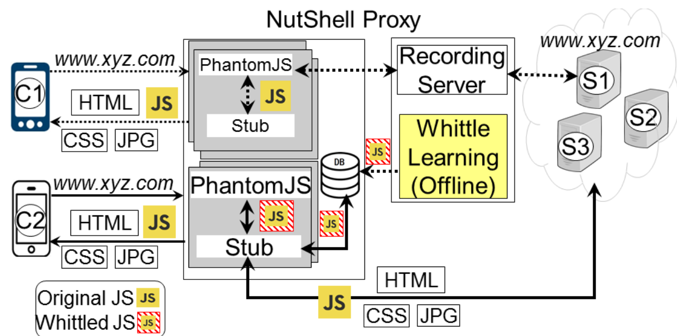


Figure 6: NutShell proxy architecture.

whittling over a 3-hour window of time while still retaining most of the benefits.

We close with two comments. First, it is important to consider reuse across users. We evaluate this further in §6.4. Second, rather than relearning over fixed time intervals, *NutShell* may be augmented to re-learn whittling based on feedback from clients – e.g., the client may report the fraction of objects needed for page load that was successfully received from the proxy. A low fraction across multiple loads is an indication that the slices must be relearnt. We do not further consider such feedback mechanisms given the effectiveness of a simple 3-hour reuse window in our evaluations above.

4 IMPLEMENTATION

Proxy implementation: *NutShell* involves extending the implementation of a proxy based on fully redundant execution to support whittling. Prior fully redundant execution schemes, Cumulus [42] and Parcel [52], use PhantomJS and Firefox respectively to parse and evaluate HTML and CSS, and execute JS. A comparison of these options indicated that PhantomJS has better scaling characteristics as it is a headless browser, which led us to employ PhantomJS in *NutShell*. To reduce computational overheads at the proxy, we disable the rendering and painting functionality which are not essential for identifying and pushing objects. We henceforth refer to this baseline fully redundant proxy as *FullRedEx*.

Fig. 6 shows our *NutShell* proxy architecture, which extends *FullRedEx* to support whittling. For each page that has undergone whittling, *NutShell* maintains (i) the JS file name; (ii) the MD5 hash of the file content; and (iii) the actual whittled version of the file. We implement a separate stub module that intercepts requests sent by the PhantomJS proxy to the server, as well as responses from the server to the proxy. When a JS file is fetched, the stub code intercepts the server response, and computes an MD5 hash of the fetched object. The index is looked up to see if there is a whittled JS file with the same hash associated with that main page. If so, the whittled version of the file is retrieved and forwarded to the PhantomJS proxy, which executes this version. The stub also pushes the unwhittled code to the client.

We use an MD5 hash rather than just the file name to (i) ensure JS content associated with that file name did not change; and (ii) to maximize the reuse of whittling in cases where the same content is fetched across runs but with slightly different URLs (a common

scenario in web downloads). In some cases, a JS file may have undergone minor changes and functions that can be whittled in the original JS code may still be whittled. As an optimization, the index stores the list of functions that can be whittled for each JS file. When the stub code receives a JS file that shares the same name as an indexed file but with a different hash, it simply whittles away functions listed in this index. While this involves some online modifications to the JS file, the overheads of such modifications is modest.

Like any execution-based proxy [6, 11, 52, 56], *NutShell* must emulate the client environment including parameters such as the User-Agent, screen width and height, viewport settings, and CSS3 media query parameters like devicePixelRatio [59] since the requested URLs may depend on these parameters. To achieve this, we use the page APIs [31] (e.g., page.settings.userAgent, and page.viewportSize) supported by PhantomJS. This is supported by most modern browsers today. The client sends these parameters when it connects to the proxy and requests the URL. The proxy dynamically creates a page object, and sets these parameters as object properties. Further, the proxy tracks the state of the objects (cache and cookies) stored at the client. This enables us to fetch the right objects and avoid transferring objects cached at the client. Like any proxy-based approach [6, 11, 52, 56], we handle HTTPS requests by assuming users trust the proxy. Such trust may be facilitated by personalized proxies [19]. Alternately, recent proposals [39, 49] that extend HTTPS to allow middleboxes to read or modify parts of the data could be adopted as part of *NutShell*.

Client implementation: We implement the client as a custom-built browser using Chromium WebView for Android 5.1.1 (rendering engine used by popular browsers). The browser accepts URLs from the user and forwards the request to the main html alone to the *NutShell* proxy. Further requests are intercepted and queued at the client. The *webview* client waits for responses pushed by the proxy, and when a response is received, matches it with a queued request if it exists. The proxy sends a flag to the client once it is done pushing all the objects required for the initial page load (§3.1). Upon receiving the flag, the client then contacts the server to obtain any remaining objects for the page load. We chose to implement a custom-built browser rather than a standard browser to facilitate the interception of client requests and to serve responses pushed by the proxy.

5 EVALUATION METHODOLOGY

Our evaluations compare *NutShell* with *FullRedEx*, a proxy that does *fully redundant execution* of all JS code (§4). We evaluate the effectiveness of *NutShell* in supporting more user requests per second by reducing JS computation at the proxy through whittling, and its ability to preserve the latency benefits of *FullRedEx*.

Test set: Since web pages change over time, and to minimize the impacts of variable server load, we used an open source record and replay tool called web-page-replay(WPR) [8] to emulate real web server. We recorded entire web pages including all constituent objects using WPR by downloading from the actual webserver(s). We then replayed the recording across all our experiments. We recorded the pages using a phone to ensure that the mobile version of the page is recorded. Note that many pages do not have separate

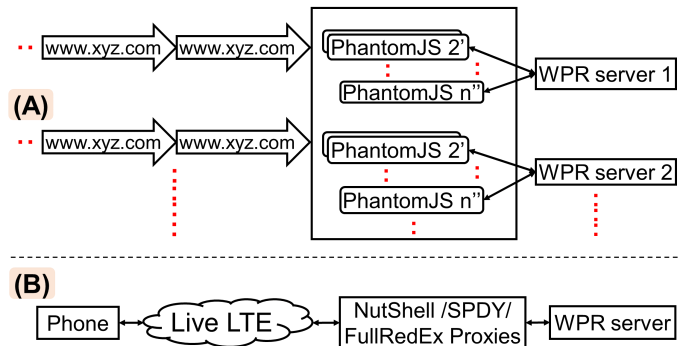


Figure 7: Experimental setup to measure (A) request throughput improvement; and (B) latency savings. For meaningful measurements, throughput experiments were performed under load, and latency measurements under a lightly loaded setup.

desktop and mobile pages, but use CSS3 media queries [59] to tailor the rendering of the page content for different devices. In either case, the right version of the page for the mobile device is recorded. For *NutShell*, we use a commonly used JS formatting tool [2] to ensure that functions (including anonymous ones) can be unambiguously identified by their line numbers to facilitate whittling.

We chose the Alexa top 100 US sites [13] for our evaluation. However, our final evaluation used 78 web pages for two reasons. First, we conducted a large number of experiments with each of our web pages, and found that ten pages did not trigger onLoad in a large fraction of experiments. Given one of our evaluation metrics depends on the onLoad event, we excluded these pages from our evaluation. Second, recall that the first step in whittling is to identify the most computationally intensive JS. While our proxy implementation is based on PhantomJS (for reasons described in §4), we are not aware of native profiler support for PhantomJS. Consequently, we employed the Chrome V8 profiler [10]. Using Chrome for profiling, and PhantomJS for slice testing and proxy implementation sometimes resulted in differences in files fetched and functions executed. Consequently, functions indicated by the profiler sometimes could not be matched to appropriate functions in the JS code.

This resulted in two issues: First, for 12 pages, none of the functions identified by the profiling step matched those actually executed by phantomJS. We excluded these pages from our analysis. Second, for all pages, a subset of functions identified by the profiler step could not be tested for whittling, limiting the amount of computation that can be saved through whittling (see §6.2). Fortunately, the issues here are not fundamental to our whittling approach. The availability of native phantomJS profiling support can both expand the set of pages we can test, as well as potentially improve the fraction of compute saved for all pages.

Measuring scaling benefits of *NutShell*: We measure the request throughput (user requests per second) under load that can be served by each of *NutShell* and *FullRedEx*. Fig. 7(A) shows our evaluation setup. Since we did not have enough mobile clients to generate sufficient load for meaningful request throughput measurements, we synthetically generated simultaneous user requests to saturate the proxy CPU by running many parallel instances of

PhantomJS. The requests from the PhantomJS instances were load balanced across five WPR servers. We made a pragmatic choice to use a commodity desktop with Intel i7 CPU @ 3.60GHz and 16 GB RAM to run the proxies, so that the number of WPR servers needed to handle the load was small. We accounted for impacts of initial ramp up and the final ramp down times by running the experiment for a sufficient duration. We ran this experiment across all the 78 web pages. We tuned the number of instances of PhantomJS and the number of requests served by each instance for each web page to ensure that the CPU was saturated for both *NutShell* and *FullRedEx*.

Setup for latency comparisons: To capture real-world impacts of cellular networks, our latency comparison experiments are done using a Google Nexus 5 phone downloading web pages over a live LTE network. In this experiment, we compared *NutShell* not only to *FullRedEx*, but also to HTTP/1.1 browser (which we refer to as *Baseline*) and to SPDY using a proxy [29] (which we refer to as SPDY). The proxy honored the default SPDY priorities (HTML > CSS & JS > images) set by our browser (Google Chrome).

We also compare *NutShell* to an approach that parses only the main HTML of a web page and pushes all objects embedded in the main HTML (which we refer to as *Push_HTMLEmbed*). We use *Push_HTMLEmbed* to generalize SPDY's server push when configured with the commonly used *embedding level 1* policy (§2). *Push_HTMLEmbed* provides an upper bound on the latency benefits of the above SPDY push approach because it also allows for pushing objects spread across multiple domains whereas a SPDY server can push objects only in its domain.

Ideally, our proxies would run in the packet core of cellular networks. Since this was not feasible, we ran an instance of each proxy on an Internet-facing server in a university campus (see Fig. 7(B)). To account for the delay from the cellular core to a typical web server, we emulate a round trip delay of 20*ms* between our proxies and the WPR server. To account for the fact that cellular networks use HTTP proxies [25], we also emulated the same delay for *Baseline* at the WPR end. We selected this 20*ms* delay based on measurements of delay when fetching the top 100 web pages from a desktop client in a university campus.

We ensured that only a single user request is served at anytime with all the schemes. We use a lightly loaded setup since our focus was on evaluating the impact on latency by reducing JS computation work at the proxy through whittling. Unlike request throughput measurements before, latency measurements require light loading to be meaningful. We expect that, in practice, these schemes would be provisioned with sufficient proxy servers to ensure small queuing delays.

We compared schemes both with respect to their Onload time (OLT) (§2), and Speed Index [7]. Speed Index is a measure of how quickly a web page's content renders on the screen. It works by calculating the completeness of a page at various points during the page load. The completeness itself is measured by comparing the distribution of colors at any instant with the final distribution after the page load. We capture a video of the page load in each of our experiments using the Android 5.1 screenrecord utility. Then we use WebPageTest's visualmetrics tool [60], to analyze the videos and generate the Speed Index metric.

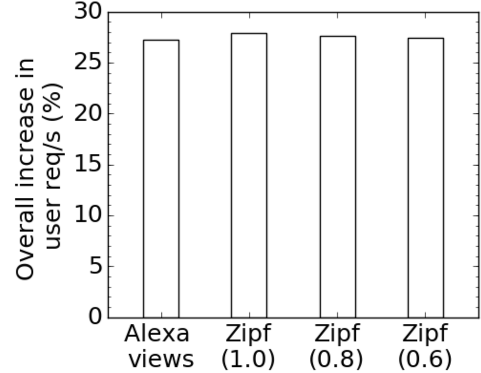


Figure 8: Overall increase in user requests per second with NutShell across page popularity models.

6 RESULTS

6.1 Scaling benefits of NutShell

We begin by presenting the effectiveness of *NutShell* in supporting more user requests per second than *FullRedEx* (§4), whose performance is representative of prior fully redundant execution schemes [42, 52].

Since a proxy would be serving multiple web pages in practice, the scaling benefits of *NutShell* depend on a combination of (i) the popularity of pages; and (ii) the savings with *NutShell* for each page. Formally, the overall benefits with *NutShell* may be computed as:

$$(\sum_i f_i \times (1/R_{if}))/(\sum_i f_i \times (1/R_{in})) \quad (1)$$

where, f_i is the fraction of requests for page i, while R_{if} and R_{in} are the number of requests per second that can be served for page i under load with *FullRedEx* and *NutShell* respectively. We obtain R_{if} and R_{in} through experiments with the setup described in Fig. 7(A).

Fig. 8 shows the increase in user requests per second with two different models for web page popularity (f_i). The first model (Alexa views) uses statistics on the number of requests to each web page estimated monthly from Alexa traffic data [9]. The second model (Zipf(α)) uses a Zipf distribution based on the Alexa rank of the page as suggested by studies on web page popularity [14, 16], where the number of accesses to a page of rank i is $1/i^{\alpha}$. We also study sensitivity to different values of the exponent α (a larger α increases the fraction of requests to the most popular page). Fig. 8 shows that across all models *NutShell* achieves fairly consistent average improvement ranging from 27.2% to 27.89%.

To further understand these benefits, Fig. 9 shows the increase in user requests per second achieved by *NutShell* over full JS execution (Y-axis) for individual pages (X-axis), sorted by the access frequency of the page. While *NutShell* provides benefits for most pages (with a 12% improvement for the median page), the benefits exceed 34% for 25% of the pages, and is as high as 100-300% for a few pages.

Further investigation shows the benefits with *NutShell* are most pronounced for pages with (i) significant JS computation, and (ii) where whittling can achieve significant reduction in such computation. For example, for *www.facebook.com*, the JS compute is significant, and whittling reduces JS computation by a factor of 2, which translates to an increase in user requests per second by

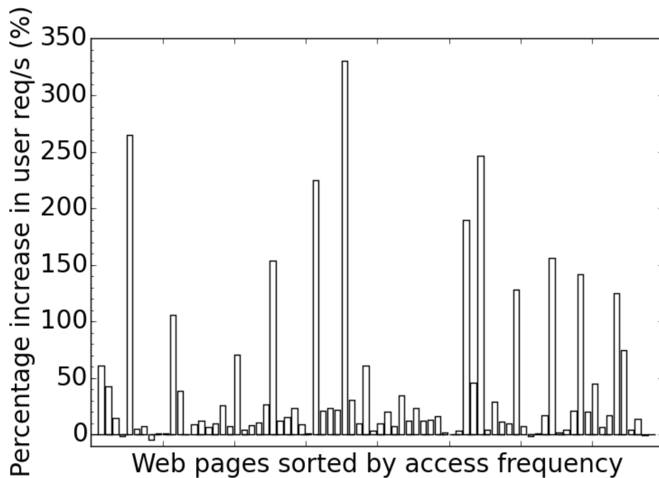


Figure 9: Increase in user requests per second with NutShell for each web page.

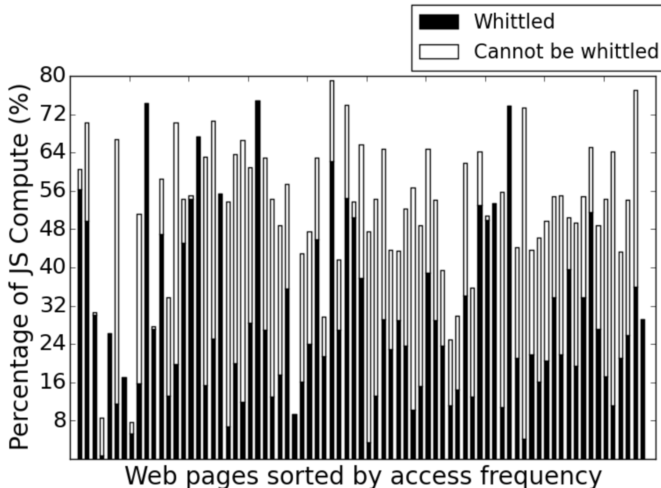


Figure 10: Percentage JS compute contributed by all the functions tested by *NutShell*, split as fraction that can and can not be whittled.

a factor of 1.43 compared to *FullRedEx*. *NutShell* achieves these benefits while still pushing all objects in the signature (§3).

A potential alternative to *NutShell* that can provide equivalent scaling benefits, is to only perform JS execution for a subset of the most popular pages. However, this approach gives up latency savings associated with proxy execution for other pages. Specifically, an analysis of Alexa web page popularities [9] indicates that the top 23 pages account for 73% of accesses of the top 100, and the top 112 pages account for 73% of accesses of the top 1000 pages. This suggests that to achieve the same 27% computation reduction that *NutShell* provides, the selective execution approach can only perform JS execution, and hence provide the associated latency benefits for 23 (112) pages. In contrast, *NutShell* can provide latency benefits for 100 (1000) pages for the same computation requirements.

6.2 Effectiveness of whittling

Fig. 10 shows the percentage of JS computation (Y-axis) that can be whittled for each page. Each bar corresponds to a page (sorted by page frequency). The lower dark and upper unshaded portions respectively correspond to the fraction of JS compute that can and cannot be whittled based on whether the associated functions were necessary for object fetches. Note that the numbers do not add up to 100% – the remainder corresponds to functions that were not tested by *NutShell* for reasons described in the next paragraph. For the left most page, the lower and upper portions are 56% and 4% respectively. *NutShell* saves more than 25% of the overall JS computation for half the pages and as much as 50-75% for 15% of the pages (thereby incurring 2X-4X lower JS computation times).

While the benefits are already substantial, these reported savings are conservative because they are based only on functions we were able to test. There are two factors that limit tested functions: (i) only heaviest functions that account for 80% of compute and at most 200 functions are tested for any page (§3); and (ii) mismatches between the browsers used for the profiling and whittling steps implied functions identified by the profiler could not be tested for whittling (§5). The first factor was relatively minor – for 85% of the pages, functions accounting for 80% of compute could be tested, while for all pages, functions accounting for at least 62% of compute could be tested. The second factor while more significant is not fundamental to our approach, and can be handled in the future through better profiler support (§5). Despite this factor, *NutShell* is still able to achieve significant savings already. For half the pages, *NutShell* can whittle over 50% of compute corresponding to the tested functions. Further, we find the overall increase in user reqs. per second by *NutShell* goes up to 40% if we only consider pages where functions corresponding to at most 20% of JS compute cannot be tested due to the second factor.

Finally, we have also considered JS computation that cannot be whittled, and investigated the extent to which dependencies required the function to be retained, though individual function testing indicated the function could be whittled. Overall, we find that savings lost due to function dependencies is not significant – *NutShell* loses JS computation savings of under 10% for 90% of the pages and at most 25% across all pages. Overall our results indicate that whittling is effective in eliminating significant fraction of the JS computation at the proxy without impacting objects fetched.

6.3 Impact of NutShell on client latency

We next present results comparing the latency of *NutShell*, with *FullRedEx* as well as other schemes – *Baseline*, SPDY and *Push_HTMLEmbed* (setup shown in Fig. 7(B)). To minimize the impact of LTE network variability, we conduct multiple rounds of experiments, with each round involving running latency experiments with all the schemes back-to-back. For each scheme, we summarize results by the median OLT and Speed Index metrices (§5) across the runs.

Fig. 11 shows *NutShell*'s speedup over each of these schemes (ratio of median OLT with a scheme to median OLT with *NutShell*). Fig. 12 shows the absolute reduction in median OLT with *NutShell* We make several points. First, *NutShell* provides a speedup of 1.7 over *Baseline* and a speedup of more than 1.5 over SPDY for half

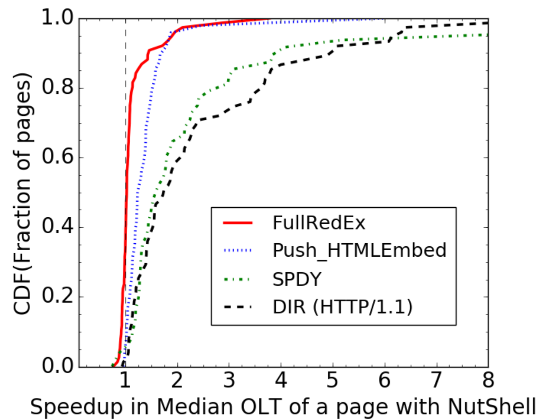


Figure 11: Speedup in median client OLT with *NutShell* compared to other schemes.

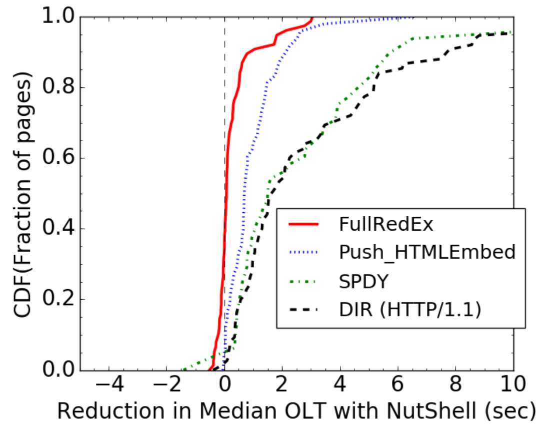


Figure 12: Reduction in median client OLT with *NutShell* compared to other schemes.

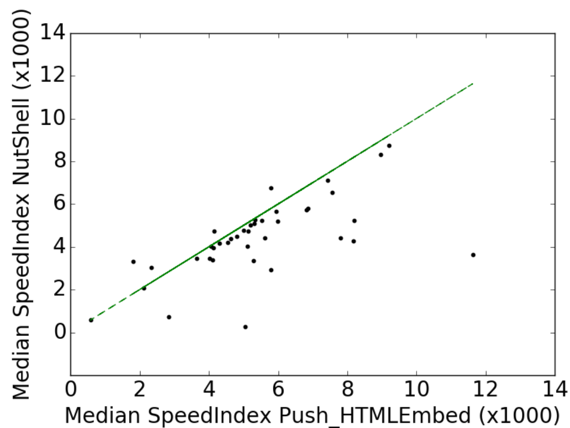


Figure 13: Median Speed Index with Push_HTMLEmbed and *NutShell*. Each dot corresponds to a page.

the pages, and latency reductions of more than 2 seconds for 45% of the pages compared to both schemes. Further analysis shows the benefits over SPDY are more pronounced for pages with deeper dependency graphs while the benefits are more limited for pages with more shallow dependency graphs. This makes sense since with SPDY the task of identifying object dependencies is still with the client.

Second, *NutShell* provides a speedup of 1.24 over *Push_HTMLEmbed* for the median page but the speedups exceed 1.5 for more than 15% of the pages. In absolute terms, this translates to latency reductions of over 1 second for 25% of the pages, with some pages seeing reductions of 6 seconds. These benefits may be attributed to *NutShell* pushing all objects as opposed to a subset.

Finally, while *NutShell* and *FullRedEx* perform comparably for the majority of pages, *NutShell* achieves speedups higher than 1.2 for 15% of the pages, and absolute latency reductions of over 1 second for 10% of the pages. The differences arise since *NutShell* lowers the OLT at the proxy since less JS computation is needed, which in turn results in objects being pushed to the client earlier. Observe that *NutShell* performs slightly worse for 34% of the pages, but only 10% of the pages see median OLT higher by 200*ms*, and no page sees median OLT higher than 515*ms*. Likewise, *NutShell* achieves latency benefits of under 500*ms* for 45% of the pages. We attribute these minor performance differences to LTE network variability.

While the results above are based on the OLT metric, we found trends generally consistent with the Speed Index metric. For example, Fig. 13 shows a scatter plot, with each point corresponding to a page, and the X-axis and Y-Axis representing the median Speed Index across the runs with *Push_HTMLEmbed* and *NutShell* respectively. A majority of points lie below the y=x line indicating *NutShell* achieves a smaller Speed Index (lower values represent better performance), and a faster visual page load from a user perspective.

Likewise, comparing *NutShell* and *FullRedEx*, the Speed Index metric results are generally consistent with OLT (not shown). *NutShell* achieves a lower Speed Index for 65% of the pages, while the Speed Index is smaller with *FullRedEx* for 35% of the pages, with the differences relatively small. Further, for most pages where *NutShell* achieves significantly lower OLT than *FullRedEx*, the Speed Index is lower as well. An exception is *www.reddit.com*, where *NutShell* achieves lower OLT but a higher Speed Index. Further analysis shows that page contains images that are shown above-the-fold, yet fetched after onLoad. Since our current *NutShell* implementation derives a signature based on objects needed for a page load event (as discussed in §3.1), *NutShell* whittles away a function responsible to fetch one of the images. Consequently, this object is not pushed by the *NutShell* proxy, and must be fetched directly by the client from the server. This issue is not inherent to whittling itself – for instance, if a signature were based on above-the-fold content, then, *NutShell* would retain necessary code, and ensure all necessary objects are pushed. Interestingly, for a few pages, notably *www.ups.com*, we found that the same phenomenon led *NutShell* to whittle code that fetched an asynchronous JS object and not push that object. In this case, *NutShell* performed better in both OLT and Speed Index by avoiding compute delays associated with the JS, since the object did not impact above-the-fold content.

Overall, these results show that beyond the primary benefit of achieving higher throughput compared to *FullRedEx*, *NutShell* can not only match the latency benefits but provide substantial latency improvements for some pages.

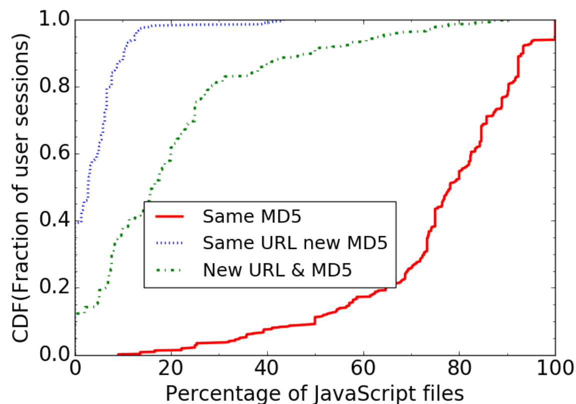


Figure 14: JavaScript code overlap across users.

6.4 Re-using whittling across users

§3.3 has shown the feasibility of reusing whittling across page loads. In this section, we study the feasibility of reusing whittling across users by analyzing common JS content among users. To this end, we conduct a user study with 14 landing pages from the Alexa top 100 pages. Each of these pages were downloaded simultaneously by 8 real users, all using the Chrome browser, but with diverse browsing profile. Further, the users corresponded to 4 different <OS,location,devicetype> settings, where the OS was Linux or Windows, location was within Purdue University, or external, and the devicetype was either a desktop or a laptop. Choosing one user as a baseline, we compare the JS files of all other users to this baseline classifying them into three categories: (i) files whose MD5 hashes match the MD5 hash of a JS file of the baseline user; (ii) files that share the same file name as the baseline user, but with a different MD5 hash; and (iii) files for which neither the MD5 hash nor file name match any JS file for the baseline user. As discussed in §4, *NutShell* can obtain full benefits with whittling for class (i) files, and a significant fraction of the benefits for class (ii) files. We repeat the analysis choosing different users as the baseline resulting in 56 user session data points for each of the 14 pages.

Fig. 14 shows a CDF of the % of JS that falls under the 3 classes across all users sessions and all pages. More than 80% of JS files have the same content across users (right-most curve) for half of the user sessions allowing full reuse of whittling. Further, the middle curve shows that for half of the user sessions, less than 18% of JS files belong to class (iii), where whittling cannot be reused. Overall, these results indicate significant common JS code across users and the potential for significant reuse of whittling across users.

6.5 Redundantly pushed data

With any redundant execution approach [42, 52] including *NutShell*, there may be differences in the URLs requested by the proxy and the client (a) if the proxy does not emulate the client (§4) faithfully, or (b) if the web page uses functions like Math.random() in JS to generate a different URL in each run. In particular, the proxy may push objects whose URLs are not requested by the client, thereby resulting in wasted bandwidth. We measure the wasted data (WD) as the percentage of bytes pushed by the proxy which are unused by the client. We report average WD across all pages weighted by the popularity of the pages (refer to §6.1). The weighted average WD is 18.4% with *FullRedEx*, and 18.3% with *NutShell* for the 'Alexa views' model (§6.1), with similar results for other models. Further investigation shows a key factor impacting the results is that PhantomJS currently does not support several HTML5 features (e.g. srcset attribute). This resulted in the proxy sometimes requesting different URLs than the mobile client even though we emulated the mobile user environment as described in §4. We believe that WD would be lower as support for these features is implemented in PhantomJS, or with an alternate browser choice for the proxy implementation. To confirm this, we repeated the above measurements using a desktop PhantomJS client and our results show that the weighted average WD is modest with both *FullRedEx* and *NutShell* (8% and 7%) respectively. *NutShell* sees slightly lower WD than *FullRedEx* because the proxy only executes the code required to fetch the signature (§3.1), which sometimes excludes URLs that vary in back-to-back runs.

Table 1: Implications of proxy execution architecture choice

	No Execution	Non-Redundant	Redundant
Prioritization	√	√	√
Compression	√	√	√
Push	Subset	All	All
Reduce Client JS	×	√	×
Complexity	Low	High	Low
Scalability bottleneck	None	Compute	Compute

7 RELATED WORK

Existing approaches to improving web page load can be classified along two dimensions: (a) proxy based execution and (b) optimizations such as content push, prioritization (controlling the order in which objects are sent) [18, 35, 41], and object compression [12, 50]. We discuss these below.

Non-redundant execution: As Table 1 shows, proxy execution based approaches themselves may be classified into (i) non-redundant execution; and (ii) redundant execution. Non-redundant proxy-based execution can reduce the compute delay at the client. Some implementations [6, 11, 62] eliminate all client side execution by getting the proxy to render the page and pushing the rendered page to the client. Though these approaches can reduce initial page load times, eliminating client execution incurs latency on user interactions (*e.g.*, mouse hover, clicks) since the JS processing of these interactions must be done in the cloud [51].

A more recent approach [56] involves partial-elimination of client-side JS code. Here, the proxy executes JS in a web page to a point and then migrates state to the client. The client continues the process from that point. Since the migrated state can become large, these approaches re-execute part (idempotent operations) of the CSS and JS code at the client. The migration of execution mid-flight from the proxy to the client makes partial elimination fairly complex. It is further complicated by issues such as modifications to the underlying JavaScript engine, browser consistency at the proxy and client, and not supporting widely-used JS constructs

such as `eval` [43] prior to page load. A recent emulation-based study [40] posits that mobile web latencies are compute-bound. Our measurements on real LTE networks with mobile clients show that the network is a significant component of latency (see Fig. 1a). Consequently, the benefit of reduced compute delay while adding network delay may be relatively small in latency-dominated cellular networks.

Content Push: The benefits of server push over basic SPDY are well known [55]. Klotski [18] does a limited form of push where only static objects (invariant across users and multiple runs) are pushed, with other objects pulled by the client. Recent works [30, 37] augment server push to ensure that the server does not push objects already in the client's cache. In contrast, we seek to solve the harder problem of identifying *all* the objects relevant to the client (including personalized content), and push those objects. With *NutShell*, we improve the scaling of redundant execution proxies to fully derive the advantages of push. Wang *et al.* [57] show the benefits of pre-loading resources of a page through speculative prefetching - we derive similar benefits through proxy-based push.

Compression, transformation and prioritization: Several popular browsers [5, 6, 12] reduce the size of data transferred by including support for data transformation and compression in the cloud. However, compression by itself does not always lower latencies [12, 50, 51]. Klotski [18] reprioritizes content so that critical content is delivered early by using a dependency structure of objects and user preferences. Incorporating user preferences may not be easy in practice. Polaris [41] proposes dynamic re-prioritization of object fetches by tracking fine-grained dependencies in Web pages. For best results with Polaris, the page has to be served from a single server. WebGaze [35] employs user gaze tracking to automatically identify critical content. Requiring users to submit to gaze tracking may not be easy in practice, and it is unclear how the approach will extend to highly personalized pages where users see varying content. That said, *NutShell* is complementary to all these above approaches, and all the mechanisms above may be readily combined with *NutShell*.

Other related work: Beyond web pages, researchers [20, 21, 36, 38, 45–47] have investigated offloading code of generic applications (*e.g.*, compute intensive face recognition applications) to the cloud, primarily to reduce computation time and save device energy. In contrast, we explore redundant execution for networking-intensive Web download. Tango [28] replicates execution at the client and the cloud, and allows either replica to lead the execution depending on which is faster during different phases of the application. Because either replica may affect user-visible content, Tango is unable to leverage approximation or to execute only a subset of JS code, which are the two key optimizations that *NutShell* employs. *NutShell*'s two-version testing has similarities to A/B testing. However, while A/B testing is typically used to measure the impact of user-visible changes on user behavior [22]. *NutShell*'s approach is an internal method to determine if a function can be whittled; end users see a single unmodified view of the Web page.

8 CONCLUSION

In this paper, we have presented *NutShell*, a proxy design that can simultaneously (i) achieve low latency over cellular networks by pushing all objects needed for a page load through redundant execution; and (ii) scale to support more simultaneous users by reducing JS computation overheads at the proxy. *NutShell* achieves the above through *whittling* – a novel technique to dynamically identify and execute only a portion of the JS code necessary to identify and push objects required for a page load. Whittling exploits the fact that approximation is acceptable at the proxy, given the client executes the full JS code. Experiments with 78 popular Alexa web sites reveal that *NutShell* sustains 27% higher user requests per second on average than *FullRedEx*. Further, by combining redundant execution and whittling, *NutShell* achieves speedups in median page load times of 1.5 compared to SPDY, and speedups of 20% compared to *FullRedEx* for 15% of the pages.

In the future, we plan to investigate ways to achieve more scaling for *NutShell* while keeping latency penalties small. A potential direction is to tune our whittling technique to eliminate functions only responsible for fetching a small number of objects. Another interesting direction is to analyze the extent of personalization in a page, and employ redundant execution for more personalized pages, and push objects based on historical accesses for less personalized pages. Finally, we also hope to further validate *NutShell* through real-world deployments.

9 ACKNOWLEDGMENTS

We thank our shepherd Matt Welsh and the anonymous reviewers for their constructive feedback and comments. This work was supported by the National Science Foundation (NSF) Award CNS-1618921. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

REFERENCES

[1] 2008. The Psychology of Web Performance. (2008). http://www.websiteoptimization.com/speed/tweak/psychology-web-performance.
[2] 2009. JavaScript beautifier. (2009). http://jsbeautifier.org.
[3] 2009. Latency - it costs you. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it. (2009).
[4] 2009. Slow pages lose users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html. (2009).
[5] 2011. Amazon Silk Split Browser Architecture. (2011). https://s3.amazonaws.com/awsdocs/AmazonSilk/latest/silk-dg.pdf.
[6] 2011. Opera Mini Architecture and JavaScript. (2011). http://dev.opera.com/articles/view/opera-mini-and-javascript/.
[7] 2012. Google Speed-index. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index. (2012).
[8] 2014. Record and play back web pages with simulated network conditions. (Oct 2014). https://github.com/chromium/web-page-replay/.
[9] 2015. Alexa Traffic Data. (2015). http://www.rank2traffic.com.
[10] 2015. Chrome devtools JavaScript CPU profiler. (2015). https://tinyurl.com/yb8uxpxl.
[11] 2017. SkyFire - Cloud based Mobile Optimization Browser (Now Opera). (2017). http://www.skyfire.com/operator-solutions/whitepapers.
[12] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, USA, 367–380. http://dl.acm.org/citation.cfm?id=2789770.2789796
[13] Alexa. 2017. (2017). Available at http://www.alexa.com/topsites.
[14] Virgílio Almeida, Azer Bestavros, Mark Crovella, and Adriana de Oliveira. 1996. Characterizing Reference Locality in the WWW. In *Proceedings of the Fourth International Conference on on Parallel and Distributed Information Systems (DIS '96)*. IEEE Computer Society, Washington, DC, USA, 92–107. http://dl.acm.org/citation.cfm?id=382006.383200

[15] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent Program Slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/2635868.2635893

[16] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, Vol. 1. 126–134 vol.1. https://doi.org/10.1109/INFCOM.1999.749260

[17] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. 2011. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 313–328.

[18] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 439–453. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/butkiewicz

[19] Ramón Cáceres, Landon Cox, Harold Lim, Amre Shakimov, and Alexander Varshavsky. 2009. Virtual Individual Servers As Privacy-preserving Proxies for Mobile Devices. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds (MobiHeld '09)*. 37–42. https://doi.org/10.1145/1592606.1592616

[20] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*. 301–314. https://doi.org/10.1145/1966445.1966473

[21] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. 2010. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys '10)*. 49–62. https://doi.org/10.1145/1814433.1814441

[22] Pete Koomen Dan Siroker. 2013. *A/B Testing: The Most Powerful Way to Turn Clicks Into Customers*. Wiley.

[23] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. 1996. Critical Slicing for Software Fault Localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '96)*. ACM, New York, NY, USA, 121–134. https://doi.org/10.1145/229000.226310

[24] KIT EATON. 2012. How One Second Could Cost Amazon 1.6 Billion In Sales. (2012). https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[25] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and K. K. Ramakrishnan. 2013. Towards a SPDY'Ier Mobile Web?. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. 303–314. https://doi.org/10.1145/2535372.2535399

[26] Stephen J. Fink and Julian Dolby. 2012. WALA-The T. J. Watson Libraries for Analysis. (2012). http://wala.sourceforge.net/.

[27] Google. 2009. Whitepaper – SPDY: An experimental protocol for a faster web. (Dec 2009). http://www.chromium.org/spdy/spdy-whitepaper.

[28] Mark S. Gordon, David Ke Hong, Peter M. Chen, Jason Flinn, Scott Mahlke, and Zhuoqing Morley Mao. 2015. Accelerating Mobile Applications Through Flip-Flop Replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*. ACM, New York, NY, USA, 137–150. https://doi.org/10.1145/2742647.2742649

[29] Ilya Grigorik. 2013. Node SPDY proxy. (2013). https://github.com/igrigorik/node-spdyproxy.

[30] Bo Han, Shuai Hao, and Feng Qian. 2015. MetaPush: Cellular-Friendly Server Push For HTTP/2. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (AllThingsCellular '15)*. ACM, New York, NY, USA, 57–62. https://doi.org/10.1145/2785971.2785972

[31] Ariya Hidayat. 2010. PhantomJS APIs. (2010). http://phantomjs.org/api/

[32] IETF. 2015. Hypertext Transfer Protocol Version 2. (June 2015). https://tools.ietf.org/html/rfc7540.

[33] Sunghwan Ihm and Vivek S Pai. 2011. Towards understanding modern web traffic. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 295–312.

[34] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

[35] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. 2017. Improving User Perceived Page Load Times Using Gaze. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/ryoo

[36] Roelof Kemp, Nicholas Palmer, Thilo Kielmann, and Henri E. Bal. 2010. Cuckoo: A Computation Offloading Framework for Smartphones.. In *MobiCASE (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering)*, Martin L. Gris and Guang Yang 0001 (Eds.), Vol. 76. Springer, 59–79. https://doi.org/10.1007/978-3-642-29336-8_4

[37] Junaid Khalid, Sharad Agarwal, Aditya Akella, and Jitendra Padhye. 2054. Improving the performance of SPDY for mobile devices (POSTER). In *Proceedings of the 16th Workshop on Mobile Computing Systems and Applications (HotMobile '15)*. ACM, New York, NY, USA.

[38] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. 2012. ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *INFOCOM, 2012 Proceedings IEEE*. 945–953. https://doi.org/10.1109/INFCOM.2012.6195845

[39] David Naylor, Kyle Schomp, Matteo Varvello, Ilias Leontiadis, Jeremy Blackburn, Diego R. López, Konstantina Papagiannaki, Pablo Rodriguez Rodriguez, and Peter Steenkiste. 2015. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 199–212. https://doi.org/10.1145/2785956.2787482

[40] Javad Nejati and Aruna Balasubramanian. 2016. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland, 1305–1315. https://doi.org/10.1145/2872427.2883014

[41] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali

[42] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 417–429. https://www.usenix.org/conference/atc15/technical-session/presentation/netravali

[43] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. 2011. The Eval That Men Do: A Large-scale Study of the Use of Eval in Javascript Applications. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP'11)*. Springer-Verlag, Berlin, Heidelberg, 52–78. http://dl.acm.org/citation.cfm?id=2032497.2032503

[44] Jim Roskind. 2013. QUIC: Design Document and Specification Rationale. (Dec 2013). http://goo.gl/p2mbcf.

[45] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. 2012. Can Offloading Save Energy for Popular Apps?. In *Proceedings of the Seventh ACM International Workshop on Mobility in the Evolving Internet Architecture (MobiArch '12)*. 3–10. https://doi.org/10.1145/2348676.2348680

[46] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K. Nurminen, Matti Kemppainen, and Pan Hui. 2012. SmartDiet: Offloading Popular Apps to Save Energy (POSTER). In *Proceedings of the 2012 ACM Conference of the Special Interest Group on Data Communications (SIGCOMM '12)*. 297–298. https://doi.org/10.1145/2342356.2342418

[47] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. 2011. The Case for VM-based Cloudlets in Mobile Computing. *Pervasive Computing, IEEE* PP, 99 (2011), 1–1. https://doi.org/10.1109/MPRV.2009.64

[48] Max Schaefer, Manu Sridharan, and Julian Dolby. 2014. Analyzing JavaScript and the Web with WALA. (2014). http://wala.sourceforge.net/files/WALAJavaScriptTutorial.pdf.

[49] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2015. Blind-Box: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 213–226. https://doi.org/10.1145/2785956.2787502

[50] Shailendra Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Ramesh Govindan. 2015. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*. ACM, New York, NY, USA, 604–616. https://doi.org/10.1145/2789168.2790128

[51] Ashiwan Sivakumar, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, Subhabrata Sen, and Oliver Spatscheck. 2014. Cloud is Not a Silver Bullet: A Case Study of Cloud-based Mobile Browsing. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile '14)*. ACM, New York, NY, USA, Article 21, 6 pages. https://doi.org/10.1145/2565585.2565601

[52] Ashiwan Sivakumar, Shankaranarayanan P N, Vijay Gopalakrishnan, Seungjoon Lee, Sanjay Rao, Subhabrata Sen, and Oliver Spatscheck. 2014. PARCEL: Proxy Assisted BRowsing in Cellular networks for Energy and Latency reduction. In *Proceedings of the Tenth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '14)*.

[53] Frank Tip. 1995. A Survey of Program Slicing Techniques. *Journal of Programming Languages* 3 (1995), 121–189.

[54] Qi Wang, Jingyu Zhou, Yuting Chen, Yizhou Zhang, and Jianjun Zhao. 2013. Extracting URLs from JavaScript via Program Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*. ACM,

New York, NY, USA, 627–630. https://doi.org/10.1145/2491411.2494583

[55] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How Speedy is SPDY?. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 387–399. http://dl.acm.org/citation.cfm?id=2616448.2616484

[56] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 109–122. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/wang

[57] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How Far Can Client-only Solutions Go for Mobile Browser Speed?. In *Proceedings of the 21st International Conference on World Wide Web (WWW '12)*. ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/2187836.2187842

[58] Mark Weiser. 1981. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*. IEEE Press, Piscataway, NJ, USA, 439–449. http://dl.acm.org/citation.cfm?id=800078.802557

[59] W3C working group. 2012. Media Queries : W3C. (2012). https://www.w3.org/TR/css3-mediaqueries/

[60] W3C working group. 2014. WPO-Foundation/visualmetrics. (2014). https://github.com/WPO-Foundation/visualmetrics

[61] Andreas Zeller. 2002. Isolating Cause-effect Chains from Computer Programs. In *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT '02/FSE-10)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/587051.587053

[62] Bo Zhao, Byung Chul Tak, and Guohong Cao. 2011. Reducing the Delay and Power Consumption of Web Browsing on Smartphones in 3G Networks. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS '11)*. 413–422. https://doi.org/10.1109/ICDCS.2011.54