Muzeel: A Dynamic Event-Driven Analyzer for JavaScript Dead Code Elimination in Mobile Web

Jesutofunmi Kupoluyi
Computer Science, NYUAD
jdk461@nyu.edu

Advised by: Yasir Zaki, Thomas Pötsch

ABSTRACT

Today's mobile web pages are becoming computationally intensive due to their focus on appearance, interactivity, and data collection in order to enhance the user experience and increase the user retention and engagement. To support the plethora of functionalities, web developers rely heavily on JavaScript, specifically large general-purpose third-party libraries. Although this tends to accelerate the development cycle, it increases the overall page complexity by bringing additional functions that might not be utilized by the page but are unnecessarily processed by the browser. In this paper, we propose *Muzeel* (which means *eliminator* in Arabic); a solution for eliminating JavaScript functions that are not used in a given web page–this is commonly referred to as *dead code*. Despite the fact that dead code is never executed, it impacts the overall performance of the web page since it is downloaded and processed by the browser. *Muzeel* extracts all of the page event listeners (upon page load), emulates user interactions using a bot that triggers each of these events, and then generates a comprehensive function call graph in order to eliminate the dead code of functions that are not called by any of these events. Our evaluation results spanning several Android mobile phones and browsers show that *Muzeel* speeds up the page load by around 30% on low-end phones, and by 25% on high-end phones under 3G network. It also reduces speed index (which is an important user experience metric) by 23% and 21% under the same network on low-end, and high-end phone, respectively. Additionally, *Muzeel* reduces the overall download size while maintaining the visual content and interactive functionality of the pages.

This report is submitted to NYUAD's capstone repository in fulfillment of NYUAD's Computer Science major graduation requirements.

Capstone Seminar, Spring 2020, Abu Dhabi, UAE
© 2020 New York University Abu Dhabi.

KEYWORDS

web simplification, superfluous javascript, javascript optimization, real user monitoring

Reference Format:
Jesutofunmi Kupoluyi. 2020. *Muzeel*: A Dynamic Event-Driven Analyzer for JavaScript Dead Code Elimination in Mobile Web. In *NYUAD Capstone Seminar Reports, Spring 2020, Abu Dhabi, UAE.* 9 pages.

1 INTRODUCTION

The reuse of existing JavaScript code is a common web development practice which speeds up the creation and the amendment of web pages, but it requires sending large JavaScript files to web browsers, even when only part of the code is actually required. In this paper, we propose to eliminate JavaScript functions that are brought to a given web page, but never used. We refer to these functions as *dead code*.

The elimination of dead code is inspired by the fact that even though an unused function is never executed, it impacts the overall performance of the page because it must be processed by the browser. The impact of JavaScript is further worsened for users who solely rely on low-end smartphones to access the web [14]. Studies show that pages require overall triple processing time on mobile devices compared to desktops [9]. While methods like script-streaming (parsing in parallel to download) and lazy parsing can reduce JavaScript processing time, only a maximum of 10% improvement of the page load time is reported [11].

New research has shown that dead code elimination has the potential to reduce the amount of first-party Javascript by 38% and third-party Javascript by up to 71% [**?**]. This, however, is an over-estimation, as concretely identifying the used/unused Javascript functions has proven difficult. Of the many difficulties associated with dead code discovery in Javascript, perhaps the most restricting has been accounting for the wide range of user interactions that could occur on a page. Given that user interactivity is a key feature that JavaScript provides in web pages, the dead code cannot be accurately identified unless all JavaScript functions that are

executed when the user interacts with the page are reported. So far research has side-stepped this challenge. [**?**] and the dynamic analyzer of [15] attempt only to eliminate code that is not required at page load. Alternatively, instead of identifying only the dead code for potential elimination, [13] removes "less-useful" functions, a process that can drastically impact the functionality of interactive pages (more than 40% loss of page functionality can be witnessed when saving 50% of the memory).

We design and implement *Muzeel*; a novel dead code elimination tool which aims to preserve user interactivity while eliminating unused code. In *Muzeel*, in addition to identifying all functions called on page load, we use browser automation to simulate all potential user events after the page loads to generate a comprehensive function call graph. Based on the generated function call graph, *Muzeel* eliminates all unused functions from the page.

In doing this, *Muzeel* aims to mitigate the impact of JavaScript on performance degradation of web pages without imposing constraints at the coding style level.

In the median page, results so far have shown a reduction in total Javascript functions by 70%, resulting in bandwidth savings of 50%, and a reduction in page load time of 1s. Qualitative analysis using pQual is actively ongoing to determine functional similarity of the pages after dead code elimination.

2 RELATED WORK

Historically, research into Javascript simplification has mainly been focused on two areas – cutting down unused modules through methods such as "tree shaking" and shrinking the amount of text in Javascript file sizes through "minification". Though minification allows Javascript files to be smaller, it may not significantly affect the processing time of Javascript in the browser[5]. This is because minified Javascript still remains relatively functionally equivalent to the original code. "Tree shaking" – removing unused Javascript modules from imported dependencies – is more ambitious in its approach and can potentially provide noticeable processing time benefits if done right [2]. Popular Javascript bundler, Webpack recently included tree-shaking (also called dead code elimination) as an optional feature during bundling[3]. Their tree-shaking process relies on static analysis to identify unused modules based on "import" statements. The breakthrough leading to its development was the emergence of Javascript ES6 which eliminated dynamic imports through the "require" statement[2]. Prior to ES6, users could import new packages at will, depending on the satisfaction of certain conditions in the program which provided a challenge to detecting unused Javascript code statically[8].

Though this has now been resolved, it was just one of many dynamic features of Javascript which were hindering static analysis. Madsen, Livshits, and Fanning, noted "JavaScript is a complex language with a number of dynamic features that are difficult or impossible to handle fully statically and hard-to understand semantic features" [12, p. 501]. Given this difficulty, dynamic analysis of Javascript modules has come to the forefront. Tools such as the Chrome Coverage API embedded in Google Chrome have been developed to help developers monitor used Javascript modules during user traces[4].

Contributing to this field of dynamic Javascript analysis, ideas have recently emerged to not just remove unused Javascript modules but to remove unused functions in imported Javascript modules, further reducing the size of Javascript files and consequently the processing time in parsing Javascript. Systems of removing unused or superfluous Javascript code have been proposed by Vazquez et al [16] as well as Goel and Steiner [10]. Both systems are largely similar and depend on a proxy server to authoritatively serve modified Javascript. Due to this constraint, Goel and Steiner target their solution to CDNs and website vendors and as such limit their solution to serving first-party Javascript[10, p. 3].

Besides the type of Javascript files that can be served, a major limitation shared with all dynamic analysis approaches, is the mechanism of attaining execution traces which capture the full range of permissible activity on the site. Vazquez et al suggested using test files provided by the developer to generate execution traces [16, p. 22]. However, developers may not know beforehand all possible interactions on the site. Further, though it is good practice, not all packages are developed with complete test coverage. Their fallback approach was to log interactions in the production environment [16, p. 22]. However, they didn't provide a scheme for generating these traces in the production environment, hence begging the question, when can we safely determine functions to be unused? Goel and Steiner propose stopping logging when the page load completes – that is when the "onloadend" event is triggered. However they concede that this greatly overestimates the number of unused functions primarily due to the fact that onclick events and other user interaction events that are triggered after page load would not be considered in such a trace [10, p. 5].

Hence, though dynamic analysis of Javascript to determine unused functions has yielded promising ideas, these ideas still need refinement to work effectively in practice.

Dead code elimination in Javascript is a very "hot topic" now [**?** , p. 402] and with the growing popularity of standalone Javascript applications, this is only set to rise. Envisioning this spike, a Javascript dead code elimination framework, Lacuna, was developed which provides an extendable

framework for developing new dead code elimination techniques, as well as combining different dead code elimination approaches [? , p. 402]. Lacuna generates a call graph of all functions in a Javascript web application [? , p. 405]. It then provides an interface for analyzers to mark which functions they identified as necessary [? , p. 406]. Lacuna then eliminates all unmarked functions on the call graph. Given that Lacuna was developed as a tool for developers/researchers [? , p. 403], it works directly on the actual source directory of a web application, on the computer in which it is hosted.

3 METHODOLOGY

We developed *Muzeel*, a dynamic JavaScript analysis tool which aims to identify and eliminate dead code in web pages while maintaining all user interactivity features on the page. To do this, *Muzeel* emulates potential user interactions with the page, by automatically triggering all events present in a web page. From this, it is a able to generate a comprehensive graph of function calls, and eliminate all functions that are not present in the graph. Unlike existing approaches, *Muzeel* runs dead code elimination autonomously without the need for "execution traces" [16] or real user interactions [10].

Muzeel is envisioned as a service offered by a CDN provider to help content owners optimize JavaScript code within their pages. JavaScript code used in today's web pages can be divided broadly into two categories: first party JavaScript, and third-party JavaScript. First party JavaScript refers to JavaScript files that are hosted within the same authoritative domain of the web page, by a CDN provider. Third-party JavaScript files refer to scripts hosted externally, outside the CDN; popular examples are Google Tag Manager, and Google Analytics [6].

Muzeel offers a flexible solution when it comes to eliminating dead code from a web page. By default, *Muzeel* eliminates dead code of first party JavaScript files, given the fact that the CDN is the authoritative entity responsible for hosting these files. For third-party JavaScript files, there are different scenarios offered by *Muzeel*, depending on the license of the JavaScript files, as well as the web page owner's preferences:

- For copyrighted JavaScript files, *Muzeel* does not perform the dead code elimination process, given that these files cannot be hosted by the CDN.
- For open-source/copylefted JavaScript files, *Muzeel* can perform the dead code elimination if the web page owner agrees to host local versions of these files.

Figure 1 shows *Muzeel*'s architecture and functioning. The dead code elimination in *Muzeel* involves a three-step process: pre-processing, dead code discovery, and dead code elimination.

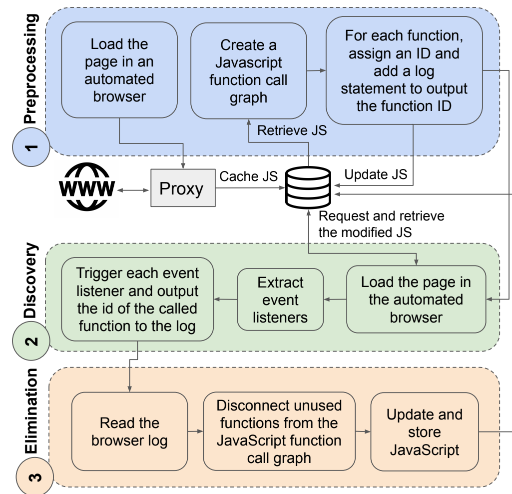


Figure 1: *Muzeel*'s architecture and functioning.

3.1 Pre-Processing

The pre-processing stage of *Muzeel* consists of multiple *phases*. In the first phase, we create an internal duplicate version of the page on which dead code elimination is carried out. This is required given that *Muzeel* modifies the JavaScript files used by a page and dynamically analyzes the modified web page in an automated environment. To create this duplicate version of the page, *Muzeel* copies and hosts the web page along with its JavaScript files in an internal CDN back-end server. In the second phase, we assign a unique ID to each JavaScript function in every JavaScript file embedded in a webpage. Each function is modified to output its assigned ID to the (browser) console when a page event that is calling that function is triggered. With the comprehensive consideration of page events, functions that are never called are labelled as *unused*.

When the identification process is completed, *Muzeel* loops through the JavaScript files used by a web page and generates an Abstract Syntax Tree (AST) per file. The AST for a JavaScript file identifies each function in the file by the name of the function as well as its start and end line numbers. *Muzeel* picks up both normal and anonymous functions – a capability that is missing in the Google Closure Compiler [10]. In *Muzeel*, each function is handled independently, regardless of whether it is nested in another function or not. In other words, functional hierarchies are not considered, given fact that *Muzeel*'s dead code elimination does not depended on function hierarchy.

Once all functions in every JavaScript file are identified, *Muzeel* constructs an edge between each function and a "base caller node" (*base*). This edge is labeled as a "constructed edge" – borrowing from the Lacuna function graph naming

convention. Figure 2 shows an example of the graph we created for a dummy webpage which imports three JavaScript libraries accounting for a total of 4 functions. Each leaf node represents an actual function and stores the URL of the Javascript file it is located in, as well as its start and end position in the file.

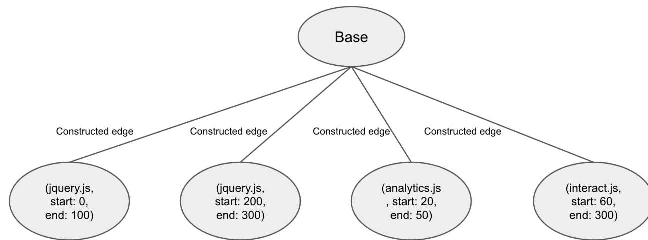


Figure 2: Function Call Graph

Our function call graph construction process is a simplification over the process described in Lacuna [15]. The major difference between the two approaches is that the function call graph we construct is of height 1 – that is to say, it is a single-level tree. Such a graph may not yield a "complete" representation of function call graph from a visual perspective – in the sense that it does not account for function hierarchy i.e. which function calls which – but it is significantly more space-efficient as less edges need to be constructed between nodes. This approach also requires less overhead in the initial construction. In Lacuna, a complete function call graph is constructed by initially connecting all functions to each other and then to the base caller node. However, as the purpose of *Muzeel* is just to identify the unused functions, a "complete" representation of the function call graph is not necessary; hence, we only connect every function to the base caller node.

When the initial call graph is created, a special console log call is added to the first line of every function, which includes the name of the file that embeds the function, as well as the function's start and end line numbers. This allows the detection of the function being called when a page event is triggered. This modification is applied to the web page's JavaScript files hosted by the CDN back-end server in order to allow serving them back during the dead code discovery process.

3.2 Dead Code Discovery

After all JavaScript functions on a web page are identified and log calls added, *Muzeel* loads the updated web page using an automated browser. We leverage the browser's built-in functionality to identify all event listeners on the page, and map these events to their corresponding elements. Given JavaScript's dynamic qualities, it is difficult to ascertain from static analysis what functions are called [15], and hence the listeners attached to objects. Therefore, leveraging a browser at this stage is crucial to identify user-based events.

After identifying and mapping the events to elements on the page, we traverse the elements of the page in depth-first order, in doing so, creating an event dependency graph based on the events mapped to these elements. For each element we encounter, we use a browser automation tool to trigger the events associated with the element. The functions called when an event is triggered are logged to the console. We monitor the console for log statements to obtain a list of functions that have been called. In the following, we discuss this and design considerations in the dead code discovery of *Muzeel*.

3.2.1 Elements Identification. In an HTML document representing a given web page, elements are structured internally using different tags. Each tag can have a unique "id", and a reference to a pre-defined "class" of attributes from the accompanying Cascading Styling Sheets (CSS) files, which set the different visual attributes for a given tag. For elements with no ancestor with an "id" or "class" attribute, we use the "body" element as the root, and use position based indexing from the body to construct the *XPath*. Using *XPath* element identification allows to identify elements across reloads.

3.2.2 Handling Events. *Muzeel* considers all commonly used JavaScript interaction-based events, including but not limited to: *mousedown*, *mouseup*, *mouseover*, *mouseout*, *keydown*, *keypress*, *keyup*, *dblclick*, *drag*, *dragstart*, and *dragend*.

Muzeel constructs an event-dependency graph and traverses this graph in a depth-first order using the *XPath*, such that a group of dependent events can be triggered in a deterministic order.

3.2.3 Hidden Elements and Attribute Changes. Web pages can contain "hidden elements", or elements which are not always visibile on the first load of the page. Additionally, elements may experience attribute changes as a result of triggering events on other elements in the page. Given that the *XPath* identifier is formed from the tag name and the "id" or "class" attributes of the elements at the first load, if an attribute change occurs, we may no longer be able to identify corresponding elements using the initially constructed *XPath*. For elements with event listeners attached, in cases of hidden elements and attribute changes, *Muzeel* adds their *XPath* to a retry queue. Then, it retries finding the element after every successful interaction with other elements. The rationale is that some interactions with another element on the page may make an element in the retry queue visible, or may reset an element's attributes to its initial state, allowing the element

in the retry queue to be discovered with its initial *XPath*. After that, if elements remain in the retry queue, we reload the page and attempt to find each of them. It is possible that elements remain in the retry queue after the end of the dead-code discovery process (See Section **??**).

3.3 Dead Code Elimination

When the above process completes, we use the browser's console logs – which contains unique identifiers of the JavaScript functions which were called – to annotate the function call graph. The functions which are left unmarked are then removed from their respective Javascript files.

Note that *Muzeel* implicitly considers the nested functions. If triggering an event leads to calling a nesting function, the nested function will also be called, and both will be reported in the browser's console, and consequently in the graph. Similarly, in the case where a nesting function is not called, all nested functions are also removed. This means that even though *Muzeel* does not preserve the hierarchy amongst the functions in the initial call graph construction, it successfully obtains a complete trace of called functions, and therefore, the functions that are not called can be identified as unused functions.

4 *MUZEEL* EVALUATION

In this section, we evaluate *Muzeel*. The evaluation revolves around: dead code elimination potential over a large dataset, performance (page timing metrics, page size, number of requests), resource utilization (CPU, memory, and battery savings), and interplay with different browsers, phone types, and networking conditions.

4.1 Methodology

Given that an actual CDN deployment is challenging, we deploy *Muzeel* using the next most realistic scenario. We assume a medium CDN provider hosting the 50,000 most popular websites [7] from Alexa's top 1M list. We then dedicate a powerful server machine (equipped with 64 cores and 1 TB of RAM) to "crawl" the landing pages of these 50,000 websites. The pages are loaded via Chrome while recording the full HTTP(S) content and headers using `mitmproxy` [1]. Next, we produce *Muzeel*-ed pages by running our dead code discovery and elimination mechanism. Finally, we setup a CDN *edge* node (10 ms latency to the user [**?**] when assuming a fast WiFi) which can serve both *original* and *Muzeel-ed* pages. This is achieved using `mitmproxy` which intercepts regular browser traffic and serves it locally, emulating the role of a CDN edge node. Testing devices are regular Android phones where mitmproxy's root CA (Certificate Authority)[1] is installed to properly handle HTTPS. Note that this step is only required for testing purposes as an actual CDN owns the certificates for the domains it serves.

The client-side testbed consists of several Android mobile devices – from low end (Xiamoi Redmi Go, Samsung J3) to high end (Samsung S10) – whose characteristics are summarized in Table **??**. For most of our tests we rely on Chrome, as it is today's most popular browser. We also experiment with Firefox, Edge, and Brave, which was chosen for its raising popularity (current 25 million monthly users [**?**]) and advanced adblocking capabilities [**?**]. The phones connect to the Internet over a fast WiFi – with a symmetric upload and download bandwidth of about 100Mbps; when needed, network throttling was used to emulate different cellular networks. We emulate three different cellular networks:

- 3G: this represents a slow cellular network with a download bandwidth of 1.6Mbps, upload bandwidth of 768 Kbps, and a round trip time of 300ms.
- LTE: this represents a moderate cellular network with a download/upload bandwidths of 12Mbps, and a round trip time of 70ms.
- LTE+: this represents a fast cellular network with a download bandwidth of 42Mbps, upload bandwidth of 25Mbps, and a round trip time of 40ms.

Each mobile device connects via USB to a Linux machine which uses the `WebPageTest` browser automation tool. This tool is used to automate both web page loads and telemetry collection, performance metrics and network requests. We focus on classic web performance metrics [**?**](FirstContentfulPaint, SpeedIndex and PageLoadTime), as well as CPU, memory, and bandwidth usage. For the Samsung J3, we also report on battery consumption measured by a power meter directly connected to the device in battery bypass [**?**]. Given that not all browsers on Android allow communication with their developer tools, which is used by `WebPageTest`, we have also developed a tool which uses the Android Debugging Bridge (adb) [**?**] to automate a browser, launch and load a webpage, while monitoring resource utilization. We then leverage `visualmetrics`[2] to extract performance timing metrics from a video of the webpage loading.

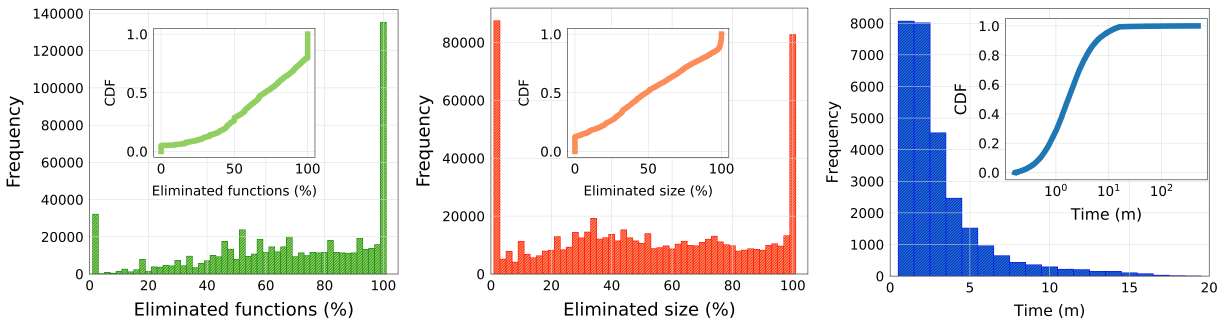
4.2 Web Pages Cloning

4.3 The Potential of Muzeel

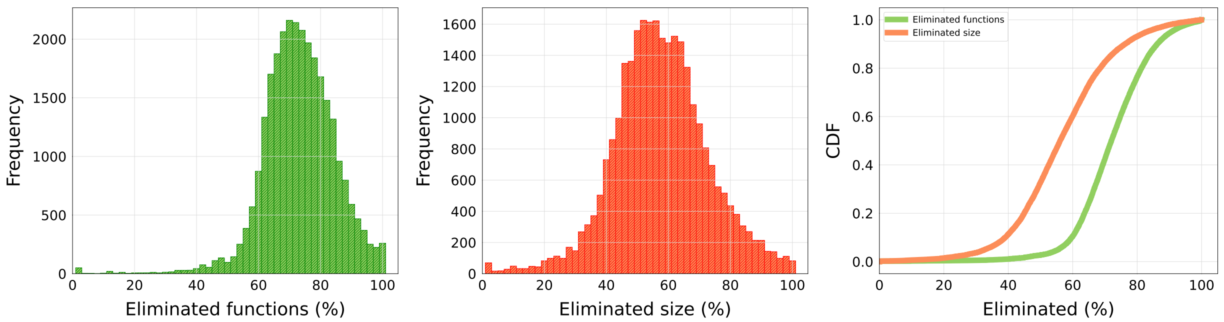
We start by studying the ability of *Muzeel* to identify and eliminate JavaScript dead code. We consider the following metrics: percentage of eliminated JavaScript functions, JavaScript size reduction, time required per page, and "running frequency", which measures how frequently *Muzeel* should run depending on how quickly web pages change.

[1] https://docs.mitmproxy.org/stable/concepts-certificates/

[2] https://github.com/WPO-Foundation/visualmetrics

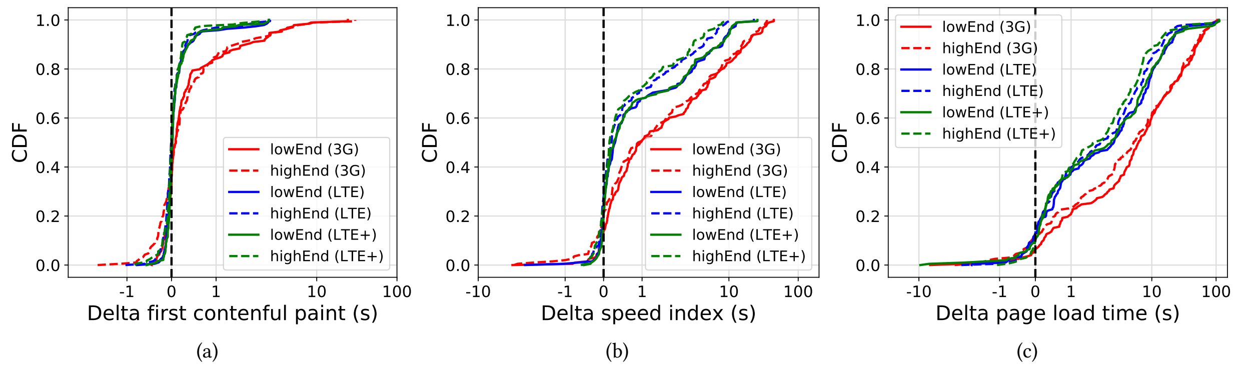


(a) Percentage of eliminated functions per JS files (b) Percentage of eliminated bytes per JS files (c) Time taken for deadcode elimination per page



(d) Percentage of eliminated functions per page (e) Percentage of eliminated bytes per page (f) CDF of eliminated functions and bytes per page

Figure 3: FIXME as discussed



(a) (b) (c)

Figure 4: Delta performance results using different networks and phone types

We used *Muzeel* to perform the dead code elimination on the top 50,000 webpages from Alexa's top 1M list. Figure 3(a),3(b) show the per-JavaScript file distribution of the eliminated percentage for both the number of functions and their bytes, respectively. The outer figures show the histograms computed over a bin size of 2, whereas the inner figures show the Cumulative Distribution Functions (CDFs). It can be seen from Figure 3(a) that about 20% of JavaScript files had nearly a 98-100% eliminated JavaScript functions[3]. The CDF also shows that, on median, the percentage of eliminated JavaScript functions per file is about 70%.

[3]This is computed as the ratio between the number of eliminated functions to the overall number of functions in the original JavaScript file.

Additionally, Figure 3(b) shows that around 80% of JavaScript files' percentage of eliminated size scatters almost uniformly across the 2-98 percentages. About 10% of the JavaScript files have a percentage of eliminated size of about 0-2% given that most of these files are small and *Muzeel* only removes the functions' bodies but keeps the functions' headers intact. On the other hand, the final 10% of the JavaScript files have an eliminated size of 98-100%. The CDF also shows that the percentage of eliminated size per JavaScript file is 50% at the median.

To understand *Muzeel*'s dead code elimination on the overall page rather than the individual JavaScript files, we computed both the percentage of the eliminated functions and the eliminated bytes on a per page bases. The results are shown in Figures 3(d) 3(e), respectively. Figure 3(d) shows a Normal distribution for the per-page percentage of eliminated JavaScript functions, with a mean around 70%. The results show that for most pages the percentage of eliminated JavaScript functions ranges between 50% and 90% (evident by the CDF shown in Figure 3(f). This is a significant deduction in the number of unused JavaScript functions which can be eliminated without impacting the pages content or functionality. Figure 3(e) shows the size reduction in bytes of the above JavaScript dead code elimination on a per page basis. The results show a similar Normal distribution with a mean of more than 55%. It can also be seen that for most of the pages the JavaScript size reduction ranges from 30% to 90% (evident by the CDF shown in Figure 3(f), which further strengthen the potential of *Muzeel*.

Finally, we assess the time complexity of *Muzeel*. We compute the time taken to perform the dead code elimination for each of the web pages. Figure 3(c) shows the histogram (and CDF) of the aforementioned time in minutes. The figure shows that for about 80% of the pages, *Muzeel* requires at most 5 minutes. Note that this duration was obtained assuming up to 6 cores used concurrently. For the remaining 20% of the pages, we measured durations of up to 20 minutes. This result suggests that *Muzeel* can be easily run each time a webpage is updated, to ensure the correctness of the JavaScript deadcode elimination. On our powerful server, by dedicating all 64 cores to the process, this process would take about 30 seconds, on median.

4.4 Muzeel Performance

In contrast to the previous evaluation that focused on highlighting the percentages of eliminated JavaScript dead code, here, we study the impact of the eliminated dead code on the overall user experience. For this evaluation, we use three Android mobile devices (see Table **??**) and 200 web pages selected from the 50,000 that we have previously cloned and *Muzeel*-ed. These pages were selected as follows. First, we consider the 1,500 most popular pages from our data-set. Next, we divided the 1,500 pages into four buckets by exponentially increasing the bucket size (doubling the bucket size every step), staring with a bucket size of 100 and ending with a bucket size of 800. Then from each bucket we uniformly chose 50 pages. We compare the performance of these 200 pages with dead code eliminated by *Muzeel* with respect to their original versions. Each page was loaded 5 times in each version, and for each metric we consider the median out of the 5 runs.

4.4.1 Network-Based Evaluation. Figure 4 shows the CDFs of the delta performance results of *Muzeel*-ed pages with respect to their original versions, in terms of the aforementioned timings metrics, using both the low-end and the high-end phones, under three emulated networks: 3G, LTE, and LTE+. For each page and metric, the delta is computed by subtracting the value of that metric measured for the *Muzeel*-ed page from the value measured for the corresponding original page. It follows that values bigger than 0 represents *Muzeel* savings, while values smaller than zero represent penalties. Experiments were conducted using Chrome.

FirstContentFulPaint (FCP) is a web quality metric capturing the first *impression* of a website, which many users often associate with what defines a webpage "fast". Figure 4(a) shows three trends, regardless of network condition and device. Some webpages (20-30%) show minimal slow down (average of few hundred ms), some exhibit no performance difference (up to 40% on LTE and LTE+), and the majority (up to 60% in 3G) show significant FCP speedups, up to several seconds. Intuitively, *Muzeel* speedups arise from JavaScripts which are saved before FCP. Given this metric is quite fast, and JavaScript tend to be loaded later in a page, it is expected to see many webpage with equivalent FCP between their original and *Muzeel*-ed version. More unexpected are the few negative results. Their explanation lies in the intricacy of the web – and the HTTP protocol itself – where removing or shrinking some objects can change the ordering of requests, by anticipating a larger object even if not contributing to a specif metric. The figure also shows much higher speedups and slowdowns when considering 3G; this is expected given that the lower bandwidth inflates the differences between the two loading strategies.

Next, we focus on SpeedIndex (SI) a web quality metric which aims at capturing the "average" user experience. Compared with FCP, Figure 4(b) shows a clear shift to the right, with now 70-80% of the pages benefiting some speedups. This happens because SI is an overall "later" metric which gives more chances to *Muzeel* to offer its savings. The same trend is also confirmed in Figure 4(c), which instead focuses on the PageLoadTime (PLT), or the time at which a browser fires the `onload` event, suggesting that all content has been

Network	Phone	PLT			SpeedIndex			Dom complete		
		%	Muzeel	Original	%	Muzeel	Original	%	Muzeel	Original
3G	LowEnd	31.5	31.7	46.3	23.4	8.5	11.1	22.7	23	29.8
3G	HighEnd	25.8	33.7	45.4	21.5	8.6	10.9	21.6	23.1	29.5
LTE	Low-End	27.3	11.2	15.5	19.5	3.8	4.8	17.9	7.9	9.6
LTE	High-End	30.1	9.4	13.5	16.3	3.3	4	16.7	6.6	7.9
LTE+	Low-End	29.2	9.7	13.7	22.1	3.4	4.3	20.9	6.2	7.8
LTE+	High-End	25.6	7.8	10.5	12.2	3	3.4	17.7	4.8	5.9

Table 1: *Muzeel*'s median results

Figure 5: CDF of bandwidth consumption across browsers: Brave, Chrome, Edge, Firefox.

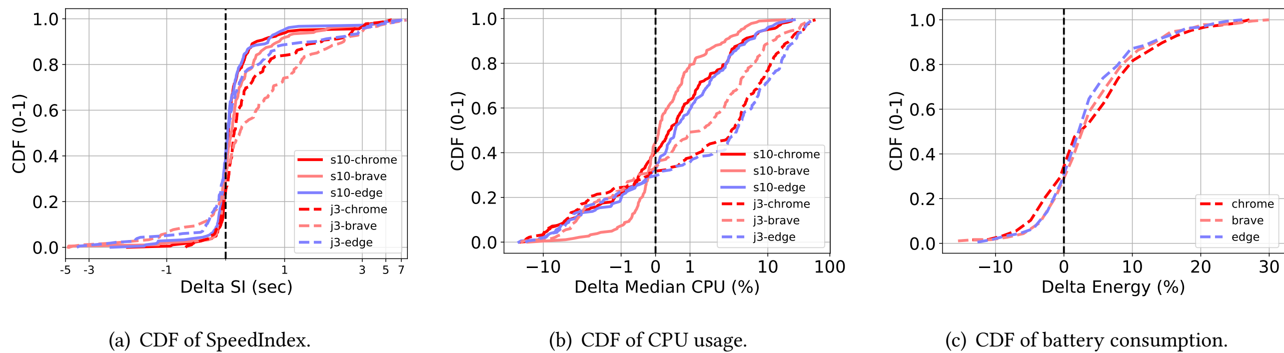
loaded. In this case, *Muzeel* offers speedups for 90-95% of the webpages. With respect to the networking conditions, both figures confirm the previous trend with much higher deltas in presence of 3G. With respect to the mobile devices, the figures show higher benefit for the low-end, likely due to a reduction in CPU usage as we will discuss later.

4.4.2 Browser-Based Evaluation. Assessed *Muzeel*'s Web performance across a variety of network conditions, we here focus on variable browsers: Chrome, Firefox, Edge, and Brave. In this evaluation, we study resource utilization (CPU, bandwidth, and battery when possible) and also report on SpeedIndex (SI). Experiments were run both on a high-end and a low-end device; differently from before, we replace the low-end device with a Samsung J3 which we have previously connected to a power meter for fine-grained battery monitoring.

Figure 5 shows the CDF of the delta bandwidth utilization (original - *Muzeel*-ed) across browsers. The figure shows, overall, very similar savings across devices for the same browser, which is expected and thus confirm correctness in the experimentation technique. The figure confirms that Chrome and Edge are very similar browsers, and indeed *Muzeel* achieves equivalent bandwidth savings on both browsers: median of about 400KB, up to multiple MBytes. Firefox is also quite similar, although providing some extra savings to a couple of websites. When considering Brave, the bandwidth savings from *Muzeel* are reduced by about 50%. This happens due to the lack of tracking and advertisement code – mostly JavaScript – which Brave removes via its integrated adblocker, thus giving less a chance to *Muzeel* to provide savings. However, even in the case of Brave *Muzeel* realizes data savings in the order of MB for 10% of the webpages. Note that Brave runs a very aggressive adblocker, and thus these numbers represent a lower bound on the expected data savings provided by *Muzeel* in presence of adblocking.

Following up from the previous result, we next investigate how fast (or slow) *Muzeel* would make the user experience across browsers and devices. We only report on SpeedIndex (SI) given that it is the web timing metric which captures the "average" end-user experience and for which the previous section has shown "average" performance improvements. Accordingly, Figure 6(a) shows the CDF of SpeedIndex per browser and device. The figure shows an overall trend similar to the previous result (see Figure 4(b)), with about 70% of the websites showing performance improvements of up to several seconds. Next, 10-20% of the pages (according to browser and device) shows minimal slow down of a maximum of 150ms, followed by a longer tail which can reach up to 5 seconds. The figure shows that Brave benefits from most performance improvements, which is counter-intuitive given the previous result on the bandwidth savings. We conjecture that this additional improvements originate from reduced load on Brave's adblocker, a complex task whose extra cost, significant especially on low-end devices, is usually amortized by bandwidth savings. In this case, *Muzeel* helps Brave's adblocker by achieving similar bandwidth savings with less computation cost from the device.

Next, we evaluate *Muzeel*'s impact on CPU consumption. We sample CPU usage once per second during a web page load, and then report the median consumption. Accordingly, Figure 6(b) shows the CDF of the (delta) median CPU utilization across browsers and devices. As above, Chrome and Edge – on a given device – achieve very similar trends. Differently from before, the figure shows a larger fraction of websites (up to 45% in the case of Brave on J3) for which *Muzeel* causes extra CPU usage, up to a 10% increase. However, note the 20-30% of these websites are within a 1% CPU increase which is just too small to be statistically significant (same holds for an even larger fraction of a 1% CPU decrease, as purposely highlighted by the x-axis). Larger CPU degradation is instead associated with websites with massive speedups (more than one second for 10-20% of the websites) where *Muzeel* compresses the overall page load in a shorter time causing a temporary burst in CPU usage. Last but not least, the figure shows overall less CPU variation, either positive or negative, in the case of Brave, regardless of the device. This happens for two reasons: 1) Brave is a lean browser with overall smaller CPU consumption than,



(a) CDF of SpeedIndex.

(b) CDF of CPU usage.

(c) CDF of battery consumption.

Figure 6

for instance, Chrome, 2) less impact due to the lacks of ads, as shown by Figure 5.

Finally, we focus on the J3 device only and comment on battery consumption. We use a power meter to derive the mAh consumed during a original and *Muzeel*-ed webpage load, and then compute their delta. Given that mAh are hard to related to actual savings/penalty, we then report the result as percentage of the battery consumption of the original version.

5 CONCLUSION

Thus far, we have developed a procedure to identify and remove unused Javascript functions from modules using dynamic analysis. Unlike previous dynamic analysis approaches which did not account for user interactivity or which required execution traces to simulate user interaction, our approach attempts to simulate all possible user interactions automatically using the event listeners attached to nodes on a page.

From a quantative standpoint, we have recorded Work is currently ongoing to evaluate the effects on the page after dead code elimination has been completed but results thus far have been promising.

Should this project be successful, we would be able to cut down not just the number of Javascript files served, but also the number of functions contained in each Javascript file. This has the potential to greatly shrink Javascript file sizes and also improve Javascript processing times on the web.

REFERENCES

[1] [n.d.]. How mitmproxy works¶. https://mitmproxy.readthedocs.io/en/v2.0.2/howmitmproxy.html
[2] [n.d.]. Reduce JavaScript Payloads with Tree Shaking | Web Fundamentals. https://developers.google.com/web/fundamentals/performance/optimizing-javascript/tree-shaking
[3] [n.d.]. Tree Shaking. https://webpack.js.org/guides/tree-shaking/
[4] [n.d.]. What's New In DevTools (Chrome 59) | Web | Google Developers. https://developers.google.com/web/updates/2017/04/devtools-release-notes
[5] [n.d.]. Why Minify JavaScript Code? https://www.cloudflare.com/learning/performance/why-minify-javascript-code/
[6] 2009. Google Analytics. https://analytics.google.com/analytics/web/. Accessed: 2021-04-23.
[7] 2021. 2020 state of the CDN industry. https://blog.intricately.com/2020-state-of-the-cdn-industry-trends-market-share-customer-size. Accessed: 2021-03-19.
[8] Ire Aderinokun. 2019. What is tree shaking and how does it work? https://bitsofco.de/what-is-tree-shaking/
[9] Houssein Djirdeh. 2019. JavaScript | 2019 | The Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2019/javascript. Accessed: 2020-01-2.
[10] Utkarsh Goel and Moritz Steiner. 2020. System to Identify and Elide Superfluous JavaScript Code for Faster Webpage Loads. *arXiv preprint arXiv:2003.07396* (2020).
[11] Marja Hölttä and Daniel Vogelheim. 2015. New JavaScript techniques for rapid page loads. https://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html. Accessed: 2021-05-4.
[12] Magnus Madsen, Benjamin Livshits, and Michael Fanning. 2013. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 499–509.
[13] Usama Naseer, Theophilus A Benson, and Ravi Netravali. 2021. WebMedic: Disentangling the Memory-Functionality Tension for the Next Billion Mobile Web Users. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*. 71–77.
[14] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-state Write Logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 249–266. https://www.usenix.org/conference/nsdi18/presentation/netravali-prophecy
[15] N. G. Obbink, I. Malavolta, G. L. Scoccia, and P. Lago. 2018. An extensible approach for taming the challenges of JavaScript dead code elimination. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 291–401. https://doi.org/10.1109/SANER.2018.8330226
[16] Hernán Ceferino Vázquez, Alexandre Bergel, S Vidal, JA Díaz Pace, and Claudia Marcos. 2019. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology* 107 (2019), 18–29.