ThorJS: A Framework for Client-Side JavaScript Optimization

Vladimir Sharkovski
Computer Science, NYUAD
vs2599@nyu.edu

Advised by: Yasir Zaki

ABSTRACT

JavaScript (JS) bloat is a significant factor in slow web performance, with many websites delivering large JS bundles that contain code rarely executed. While existing solutions for JS "dead code" elimination can be effective, they all require server-side support, which limits adoption and contributes to web ossification. In this capstone paper, we introduce ThorJS, an innovative client-side technique for JS dead code elimination. ThorJS leverages HTTP range queries, allowing browsers to fetch only the essential segments of JS code required for specific user interactions. This method requires no server-side participation, enabling seamless adoption across diverse websites. To assess the feasibility of this approach, we first analyze the top one million websites, identifying near one million unique domains serving JS; for each domain we evaluate their support for HTTP range queries, including single and multi-range requests. We then implement ThorJS in a lightweight, Chromium-based browser called Thorium (which inspired the name ThorJS) and benchmark eleven representative websites to evaluate ThorJS improvements in user browsing experience and resource efficiency.

KEYWORDS

range requests, javascript, web browsers, internet

Reference Format:
Vladimir Sharkovski. 2024. ThorJS: A Framework for Client-Side JavaScript Optimization. In *NYUAD Capstone Project 1 Reports, Fall 2024, Abu Dhabi, UAE*. 7 pages.

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

جامعة نيويورك أبوظبي
NYU ABU DHABI

Capstone Project 1, Fall 2024, Abu Dhabi, UAE

1 INTRODUCTION

JavaScript (JS) has become the backbone of modern web applications, enabling rich interactivity and complex client-side functionalities. Many websites include large, monolithic JS files, even though only a fraction of the code might be necessary. This pervasive JS *bloat* leads to high CPU and battery usage, and slow page load times, particularly affecting users on mobile devices or slow networks. Despite advances in JS minification [5, 8, 14] and dead code elimination [2, 12, 15–18], these solutions are limited in scope and require active server-side participation, creating a barrier to adoption.

This paper introduces *ThorJS*, a novel client-side method for JS dead code elimination, which avoids the need for server-side participation. ThorJS's intuition is to leverage HTTP range queries to enable clients to request only the essential portions of JS code required for a specific interaction. This method allows users to load minimal JS tailored to their needs, significantly reducing the overhead of downloading and executing unnecessary code. By relying solely on client-initiated code requests, ThorJS bypasses the need for servers to opt in or adjust configurations, reducing Web ossification.

First we investigate the support of HTTP range queries across JS serving domains in the Internet. We conduct a comprehensive analysis of the top one million websites –as per the HTTP archive on 10/01/2024 [3]– from which we extract roughly one million unique domains that serve JS content. We then test the support of HTTP range queries, both single and multi-ranges, requesting a JS file per each serving domain. We find that support is widespread, with a quarter of domains supporting multi-range and single-range queries and another quarter supporting only single-range queries.

Then, we implement ThorJS within Thorium, a simplified Chromium-based browser. We benchmark ThorJS on 11 websites, where we measure key performance indicators, including Speed Index and Page Load Time. We find that our implementation of ThorJS instead has a negative performance impact, and consider possible reasons for that.

2 BACKGROUND AND RELATED WORK

JavaScript (JS) is one of the key contributors to (high) webpage load times, making webpages feel sluggish especially on low-end devices [20]. With the goal of speeding up and compressing webpages, the research community has invested significant efforts in JS dead code elimination. Lacuna [17] pioneered JS dead code elimination at a functional level, combining static and dynamic analysis within a generic framework. Two recent methods [12, 18] also focus on JS dead code elimination through dynamic analysis. [18] uses unit tests to identify unused code, while [12] relies on real user monitoring (RUM) [2] to track user interactions. WebMedic [16] approaches dead code elimination by removing low-priority functions to reduce memory usage on low-end devices. This strategy yields an average 50% memory savings but compromises up to 40% of page functionality. Muzeel [15] improves upon these works by handling the dynamic nature of JS while introducing user-page interaction, achieving high visual appearance of webpages and interactive functionality (similarity score of above 90%).

Despite their differences, all the above approaches share one common limitation: they only work on the server-side. That is, they require existing servers to adopt them and directly serve more optimized JS code. By contrast, ThorJS achieves dead code elimination without any server side support by directly requesting only the portions of required JS code. This is realized via HTTP range requests, which allow clients to request specific parts (ranges) of a resource rather than downloading the entire file. By including a "Range" header in the HTTP request, a client can specify which byte ranges it needs, such as "Range: bytes=0-1024", to retrieve only the first 1 KB of a file. If the server supports range requests, it will respond with a "206 Partial Content" status and send back only the requested range, rather than the entire file. Multi-range requests allow clients to ask for multiple, non-contiguous segments in a single request, such as "Range: bytes=0-1024, 2048-3072", enabling efficient access to different parts of a file.

No previous work has yet investigated the current support for HTTP range requests in the Internet. While all major server engines (Apache, Nginx, etc.) support HTTP range requests, system administrators might choose to disable them, despite the lack of clear incentive to disable such feature to the best of our knowledge.

3 IDEA VALIDATION

ThorJS is founded on the idea that HTTP range requests can be used to enable browsers to only request *essential* portions of JS code required for a specific interaction. In this section, we validate the presence or absence of support of HTTP range queries across JavaScript (JS) serving domains in the Internet.

3.1 Range Request Support

Methodology. We use BigQuery to create a dataset from the HTTP Archive [3], containing the URLs of all JS files embedded in pages within the top one million sites by popularity. This dataset is generated on October 1, 2024, and spans 61,466,389 individual URLs from 999,733 unique JS-serving domains. On average, each domain hosts 61.6 unique JS files/URLs.

Our next step is to evaluate each domain for support of HTTP multi-range and single-range requests. We select a random JS URL from each domain cluster and send a HEAD request specifying a multi-range (e.g. bytes 0-1 and 10-11). The response headers are then analyzed to check for multi-range support, indicated by the presence of an "Accept-Ranges: bytes" header. If multi-range support is absent, we proceed to test single-range requests before moving to the next domain. An assumption derived from smaller-scale testing is that if a domain supports multi-range requests, it already supports single-range requests.

The evaluation was done from the NYUAD campus. However, given the prevalence of Content Delivery Networks (CDNs) among these domains, it would be beneficial to emulate requests from varied geographic locations, increasing the chance of testing different CDN servers, which might exhibit different behaviors. This can be achieved by changing the public IP address of our testing client with each complete cycle of 999,733 domains, for example with a Virtual Private Network (VPN) [1].

Executing the evaluation took 24 hours, about 12 seconds per domain. A timeout of 10 seconds was used for requests. A single machine with a single testing client was used. The testing client is multi-threaded, simultaneously testing 32 different domains at any moment; 32 threads were found to perform better despite the machine being used having 64 logical cores.

Results. We find that of the 997,733 domains, 229,725 (23.0%) support multi-range requests and single-range requests, 233,642 (23.4%) support only single-range requests, and 534,366 (53.6%) support neither. With nearly half of the domains supporting either kind of range requests, we see that ThorJS can be applied to a large fraction of the internet being visited today.

4 THORJS

This section describes ThorJS, a client side approach to JS dead code elimination. At a high level (see Figure 1) ThorJS

[1] Unfortunately, this was not attempted for the capstone due to time and resource constraints.

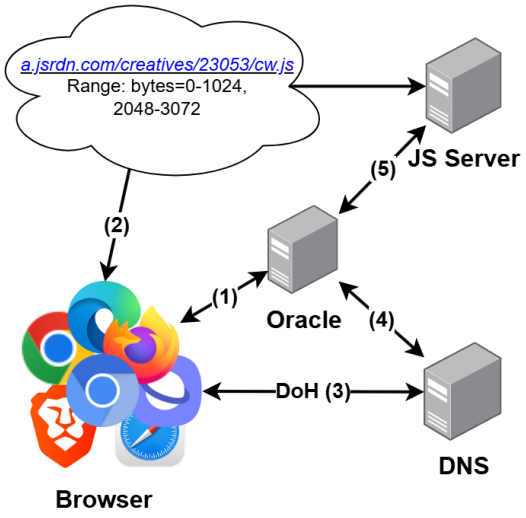

Figure 1: ThorJS's architecture and workflow.

consists of a modified Web browser which relies on HTTP range requests to retrieve only the portion of JS files needed, thus achieving code elimination (2). This information, i.e. required ranges per JS file and domain, is offered by an *oracle* (1). The oracle's role is to run classic JS dead code elimination techniques against a set of popular websites (5), in order to maintain up-to-date information of which code portions can be suppressed, in the form of *range headers*.

4.1 Oracle

Although the oracle could also be implemented in the browser, we instead opt for it to be operated by an Internet Service Provider (ISP). While this approach may introduce some privacy considerations, as discussed below, it is an overall more efficient approach as each browser does not need to re-run the same dead code elimination algorithm when visiting the same website. Further, even the first visit to a website can already leverage the benefits of JS dead code elimination.

In the example shown in Figure 1, the oracle is operated by an ISP in collaboration with the ISP-provided Domain Name System (DNS) resolver. This approach offers two key advantages. First, DNS inherently collects statistics on popular websites within a given network or geographic region. Second, the oracle does not introduce any additional privacy concerns beyond those already present in DNS traffic. Alternatively, a centralized oracle model is also feasible, similar to the structure of cloud DNS providers like Google [13] and Cloudflare [9].

Accordingly, DNS traffic (3) is used to drive the oracle (4) with information on which websites are currently popular among the users served by a given DNS, i.e. a given region. The oracle runs one or several JS dead code elimination strategies (5), e.g. Muzeel or Lacuna, and populates a "range file", which contains for each known JS URL the range headers to be used, if any. This file is maintained over time with a proper versioning system so that only partial increments/variations need to be sent to a requesting browser (1).

4.2 Browser Implementation

Thorium [11] is an open-source Chromium-based web browser aiming to provide performance improvements over vanilla Chromium. We decided to implement ThorJS in Thorium due to their helpful documentation in terms of set-up, development, and building. However, our implementation may be applied directly to Chromium or other Chromium forks, and we shall refer to Chromium here as an umbrella term for these.

Range File Management. At startup, and at hourly frequency, the browser communicates with the oracle to check whether a new version of the range file should be downloaded. In a more comprehensive implementation, given that small changes to this file are expected, delta updates may be used together with existing caching mechanisms in the browser.

It is necessary to decide where to store and how to store the range file in the browser, and how to access it when requests are made. Depending on where the range file is stored, loading a website may initiate many requests for JS files, which all necessitate reading the range file. This is not straightforward because Chromium has a multi-process architecture, with one *browser* process and one or more rendering processes (*renderers*) [7]. The browser process is always alive, while renderers may be created or killed as tabs are opened or closed, with one renderer per tab.

We considered two potential solutions. The first is to load the range file in the browser's memory and have renderers communicate with the browser through inter-process communication (IPC). Whenever a website is loaded, for each JS file, the renderer queries for potential range data for that file. While this leads to dozens of IPC messages per page load, IPC messages are fast in modern systems and would probably not be the bottleneck in the implementation. Moreover, it is memory-efficient because range data is stored only once. An alternative approach is to replicate the range data and keep it in-memory within each renderer. This increases the memory footprint of the browser, but does not introduce many IPC messages when loading webpages.

While we believe that the first solution is more sensible, it was difficult to implement in time for the capstone, so we opted for the second one, visualized in Figure 2. In the

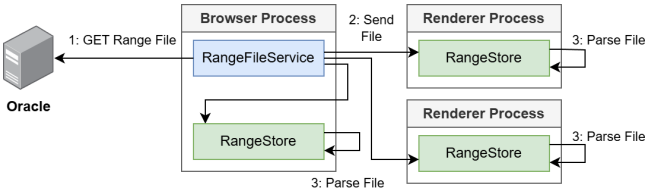

Figure 2: Workflow associated with the range file retrieval and processing.

browser process, an entity called the *RangeFileService* handles the retrieval of the range file, either by sending a GET request to the oracle or by accessing it from the browser cache. Next, it sends the file to all *RangeStores* through IPC; there is one RangeStore in the browser process and one in each renderer, and the file is also sent to any renderers created in the future. Next, each RangeStore parses the file, creating an in-memory mapping of domain and file URL to *Range header* [2].

Request Interception. To intercept requests, we use a functionality provided by Chromium called *throttles*, which can intercept HTTP requests at different points in their lifetime [1]. Chromium uses many throttles for its features, instances of which are created once per request. We implement our own throttle, which intercepts requests at two points. Before the request is sent, the throttle attempts to convert it into a multipart byte-range request [3]. Afterwards, when the response starts being received, depending on the response code, as discussed below, the throttle "intercepts" the response body and forwards only the necessary parts, using the byte ranges knowledge. By forwarding only the necessary parts of the response body, we reduce the load on the JavaScript engine afterwards.

Figure 3 summarizes the throttle operations, assuming a RangeStore has been initialized in the process. For each request, after Chromium initializes its own existing throttles and adds them to a list, we add our own throttle at the end. This attempts to avoid unexpected behavior as other throttles may also be modifying the request. When the request is about to be sent, our throttle ensures that the request is a GET, and that it is not already a range request, i.e. it does not have the "Range" header [4]. The throttle determines the domain

[2]A similar option would have been to parse the file once and send the in-memory mapping through IPC, but the parsing is simple, and it was not straightforward to send the data structure through the Chromium IPC system.

[3]This can only be done when the domain supports multipart requests, which we know from the range file. If the domain only supported single-part range requests, another useful functionality could be applied, where the request is split into multiple single-part range requests; however, this was not implemented in the capstone due to time constraints.

[4]It would also be beneficial to filter for only JS files at this point, but there is not enough information to know whether the request is for a JS file.

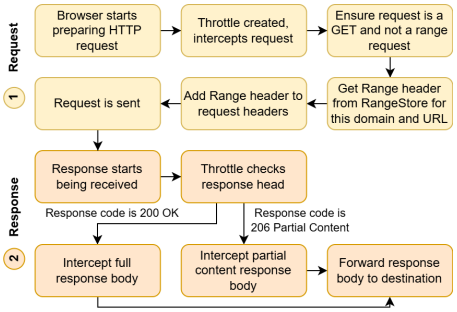

Figure 3: Workflow associated with intercepted JS requests and responses.

hosting the JS being requested, and queries the RangeStore to find the Range header associated with this resource. If the header does not exist, we stop intercepting and let the request continue unmodified. Otherwise, the request is sent to the network.

Response Interception. As a response starts being received, the throttle is notified. The throttle only has access to the response header at this point, not the body. The throttle takes action based on the response code; if the response code is 200 (OK/Success), the whole JS file is being returned despite the throttle having converted the request to a multipart byte-range request [5]. This suggests lack of range request support at the server, thus a mismatch with respect to the information contained in the RangeStore [6]. The throttle initializes a helper object to intercept the response body and forward only the relevant parts, thus still realizing *local* benefits from partial JS usage (but not download). How this process interacts with the browser caches is discussed in the next section.

If the response code is 206 (Partial Content), the range request has been respected. Due to how range requests are specified, the response body will have additional metadata inserted between the parts requested by the ranges — see Figure 4. To handle this, the throttle creates another helper object which will parse the response body, removing the metadata and forwarding to the renderer only the file content as specified with the byte ranges.

Cache Interaction. It is important to consider how our implementation interacts with the browser's caching mechanisms. Chromium has an *HTTP Cache* module which determines whether to serve HTTP requests from the lower-level disk cache or from the network [6]. There is also a different

Methods such as checking if the URL ends with a *.js* would disqualify many JS files, and we do not know the *Content-Type* header of the future response.

[5]This is an automatic fallback mechanism observed by testing; we did not encounter situations where a "not supported" response was returned.

[6]In a more comprehensive implementation, this information may be propagated back to the oracle as it may be useful for updating the range file.

```
--4c474222fbb95bc8
Content-type: application/javascript
Content-range: bytes 0-13/14244

!function(e,t)
--4c474222fbb95bc8
Content-type: application/javascript
Content-range: bytes 186-201/14244

(this,function()
--4c474222fbb95bc8
Content-type: application/javascript
Content-range: bytes 14210-14243/14244

);
//# sourceMappingURL=aos.js.map
--4c474222fbb95bc8--
```

Figure 4: An example response for a multipart byte-range request. The parts which were requested are in black, while the additional metadata is in blue.

in-memory cache in each renderer. Our throttle sits in the network stack and interacts with the HTTP cache, while the in-memory cache exists at a higher level above the network stack. Therefore the in-memory cache was expected to perform as normal, which testing confirmed. Now we discuss the throttle's interactions with the HTTP cache.

From our tests, we found that modifying requests for JS files, as we did with our throttle, led the requests to never be served from the disk cache, but to instead be sent through the network. A possible reason is the cache's *sparse entries* mechanism. Besides normal entries which store entire resources, the cache supports sparse entries, which allow storing only parts of resources, in cases where the whole resource has not been requested. We assume sparse entries are used for any requests our throttle intercepts. However, the browser already uses byte-range requests as a mechanism to create sparse entries. Therefore, as our throttle is also converting requests to byte-range requests at an unexpected point in the request workflow (from the unmodified browser's perspective), this may be causing the cache to behave unexpectedly. For example, our requests may be failing cache validation, leading to the browser always issuing network requests instead.

For the capstone course, due to time constraints and the complicated nature of the HTTP cache module, we were unable to determine the exact reason for why our modified requests were not being served from the disk cache, as well as modify our implementation to address that. Therefore our performance evaluation in Section 5 disabled the HTTP cache.

5 PERFORMANCE EVALUATION

This section benchmarks ThorJS with respect to user QoE measured with web performance metrics. We focus on Muzeel [15] which is, to the best of our knowledge, the most advanced solution for JS deadcode elimination. Nevertheless, ThorJS is a generic approach and can be used with any existing or future solutions.

5.1 Methodology

We start by identifying a representative set of testing webpages. Using the dataset described in Section 3, we obtain a random sample of 11 webpages [7]. We then run Muzeel on these webpages to identify the byte ranges for all JS files, and use Muzeel's output data to construct the range file.

We use `webpagetest` [21] to automate the loading of these webpages via a modified Thorium browser with ThorJS capabilites. We clear the browser cache between each load for reasons discussed in Section 4.2. Each webpage is loaded three times using the following network configurations: 1) **Slow**: 20 Mbps downlink/uplink rates with 100 ms RTT, 2) **Medium**: 50 Mbps downlink/uplink rates with 50 ms RTT, and 3) **Fast**: 100 Mbps downlink/uplink rates with 20 ms RTT. A low-end mobile device (Xiaomi Redmi Go with a Quad-core 1.4 GHz CPU, and 1GB RAM) is used to load each webpage using the following configurations [8]: *Classic* where ThorJS is simply disabled and thus legacy Thorium is used; *Multi* which only leverage multi-range HTTP requests; and *Local* where regular JS code is requested, but local splitting is used to forward only the amount of JS code required.

As web performance metrics, we measure First Contentful Paint (FCP) [19], which is a user-centric metric measuring perceived load speed as it marks the first point in the page load timeline. Next, Speed Index (SI) [4] which measures how quickly a website's content is visually displayed during load. Finally, Page Load Time (PLT) [10] which measures the amount of time it takes for a webpage to fully load.

5.2 Results

Tables 1, 2, and 3 show the results for the average FCP, SI, and PLT metrics across all pages, when varying the network and browser configurations. The results generally show a performance decrease (instead of an increase) for all metrics when using the Local or Multi configuration of ThorJS. For FCP, the most significant improvements are seen over Slow networks, where Local and Multi respectively provide a 27% and 12% improvement over Classic. For SI, only Local for

[7] We would have opted for a larger number of webpages, but this was not possible for the capstone due to time constraints and the fact that executing Muzeel was slow with our resources.

[8] Targeting desktop would also be useful, but primarily focusing on mobile devices is in accordance with our motivation as described in Section 1.

	Classic	Local	Multi
Slow	5.44	3.99 (-26.6%)	4.77 (-12.3%)
Medium	4.55	4.55 (-0.1%)	4.22 (-7.4%)
Fast	4.06	4.48 (+10.3%)	6.66 (+64.1%)

Table 1: Results for First Contentful Paint. Cells are average FCP values, in seconds. The numbers in parentheses are comparisons to the baseline *Classic*.

	Classic	Local	Multi
Slow	11.50	11.39 (-1.0%)	11.71 (+1.8%)
Medium	12.31	10.58 (-14.0%)	11.78 (-4.2%)
Fast	9.93	10.34 (+4.1%)	16.66 (+67.7%)

Table 2: Results for Speed Index. Cells are average SI values, in seconds. The numbers in parentheses are comparisons to the baseline *Classic*.

	Classic	Local	Multi
Slow	28.25	31.08 (+10.0%)	32.13 (+13.7%)
Medium	27.02	28.18 (+4.3%)	32.11 (+18.8%)
Fast	26.89	26.21 (-2.5%)	47.74 (+77.6%)

Table 3: Results for Page Load Time. Cells are average PLT values, in seconds. The numbers in parentheses are comparisons to the baseline *Classic*.

Medium networks shows a 14% improvement, and for PLT, only Local for Fast networks shows a 2.5% improvement. For all other cases of FCP, SI, and PLT, the Classic (unmodified) browser performs best. Another noticeable result is that Multi performs very badly over Fast networks, being more than 64% slower than Classic on all metrics.

For the cases where there is improvement from Local and Multi, it is possible that ThorJS is performing as expected, in that both Local and Multi are passing less JS code to the browser's JavaScript engine, and this ultimately speeds up the loading. However, these results are not statistically significant, as only 11 websites were tested. In most cases, the performance of Local and Multi is worse than Classic.

A possible reason for Local's worse performance is with how it is implemented (see Section 4.2). In the browser, the throttle is introduced as an additional component in the middle of the flow of execution of requests for JS files. The throttle forwards only parts of the response body to the rest of the workflow, but this forwarding necessitates additional copying of the response body data: a full copy for the throttle to receive it and then additional copies of each part which will be forwarded. These copies of data, which require underlying memory allocations and copying, together with the mechanisms Chromium to redirect response data to our throttle, may be slowing down the performance, especially for large files. To mitigate this performance impact, it might be necessary to implement this functionality in a better way, possibly deeper in the browser's network stack, as well as to modify other browser components in accordance with the optimization.

The worse performance of Multi can also be caused by similar reasons to the one of Local. In this case, the implementation is slower by necessity, as Multi has to parse the contents of the response body before forwarding certain parts, while Local immediately knows the byte ranges to forward without having to do any parsing, due to having access to the Range header. However, there is another possible reason why Multi is performing badly, especially in the case of fast networks. Multi sends multipart byte-range requests, which ask the JS server (see Figure 1) to process the JS resource on its own end before sending a response. This additional processing may take an unexpectedly large amount of time compared to the case of Classic, where the JS server simply returns the JS file. Moreover, in the case of Multi, the byte range requests are bypassing any type of caching outside the browser, such as server-side caching and CDNs. All these effects are most clearly visible over a fast network, as less time is spent over the network, emphasizing the performance of the browser and JS server.

6 CONCLUSION

In this capstone paper, we introduced ThorJS, a client-side approach for JS dead code elimination that leverages HTTP range queries to fetch only the necessary portions of JS files when loading websites. ThorJS is designed to avoid server-side dependencies in order to enable easier adoption across the internet and to avoid web ossification.

Our analysis of the top one million websites revealed that HTTP range query support is prevalent, with approximately 50% of JS-serving domains supporting single or multi-range queries. This highlights that ThorJS may be a useful client-side solution for JS optimization. However, our performance evaluation of ThorJS led to mixed results. While isolated cases showed minuscule improvements in metrics such as First Contentful Paint, Speed Index, and Page Load Time, in most cases, ThorJS performed worse than an unmodified browser.

We then discuss possible reasons for ThorJS's worse-than-expected performance, namely limitations in the browser implementation and the possible effect of bypassing caching mechanisms outside the browser. We also emphasize how the results of the evaluation should not be taken for granted, as only a very small number of websites could be tested.

Despite the current limitations, ThorJS opens avenues for addressing JS bloat from the client side. Future work should

focus on a more comprehensive and efficient browser implementation, understanding the effects of all types of caches –inside the browser and outside– on range requests, and executing a performance evaluation with a larger scope. These improvements may lead to innovative ways to handle JS bloat and its effects on web performance and user experience.

REFERENCES

[1] John Abd-El-Malek. 2016. Network Service in Chrome. https://docs.google.com/document/d/1wAHLw9h7gGuqJNCgG1mP1BmLtCGfZ2pys-PdZQ1vg7M/edit. Accessed: 2024-12-03.

[2] Akamai. 2020. RUM JavaScript. https://learn.akamai.com/en-us/webhelp/ion/real-user-monitoring-guide/GUID-E6BE104B-2EB6-463B-A0C8-7A3D0E746446.html.

[3] HTTP Archive. 2024. HTTP Archive Dataset. https://httparchive.org/. Hosted on Google BigQuery. Accessed: 2024-11-15.

[4] Cody Arsenault. [n.d.]. Speed Index Explained - Another Way to Measure Web Performance. https://www.keycdn.com/blog/speed-index.

[5] Mihai Bazon. 2012. UglifyJS. http://lisperator.net/uglifyjs/. Accessed: 2020-05-01.

[6] Chromium. 2024. HTTP Cache Documentation. https://www.chromium.org/developers/design-documents/network-stack/http-cache/. Accessed: 2024-12-04.

[7] Chromium. 2024. Multi-process Architecture. https://www.chromium.org/developers/design-documents/multi-process-architecture/. Accessed: 2024-12-06.

[8] CircleCell. 2011. JSCompress - The JavaScript Compression Tool. https://jscompress.com/. Accessed: 2020-05-01.

[9] Cloudflare. 2024. Cloudflare DNS. https://www.cloudflare.com/learning/dns/what-is-1.1.1.1. Accessed: 2024-12-16.

[10] MDN Web Docs. 2024. Page load time. https://developer.mozilla.org/en-US/docs/Glossary/Page_load_time. Accessed: 2024-12-06.

[11] Alexander Frick. 2024. Thorium Browser. https://thorium.rocks/. Accessed: 2024-12-03.

[12] Utkarsh Goel and Moritz Steiner. 2020. System to Identify and Elide Superfluous JavaScript Code for Faster Webpage Loads. *arXiv preprint arXiv:2003.07396* (2020).

[13] Google. 2024. Google Public DNS. https://developers.google.com/speed/public-dns. Accessed: 2024-12-16.

[14] Kayce Basques. 2018. Find Unused JavaScript And CSS With The Coverage Tab. https://developer.chrome.com/docs/devtools/coverage/.

[15] Jesutofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Russell Coke, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. 2022. Muzeel: Assessing the impact of JavaScript dead code elimination on mobile web performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 335–348.

[16] 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.

[17] Niels Groot Obbink, Ivano Malavolta, Gian Luca Scoccia, and Patricia 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)*. IEEE, 291–401.

[18] 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.

[19] Philip Walton. 2024. First Contentful Paint (FCP). https://web.dev/articles/fcp. Accessed: 2024-12-06.

[20] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao

[21] WebPageTest. 2024. WebPageTest. https://www.webpagetest.org/. Accessed: 2024-12-06.