Revisiting Browser Performance Benchmarking From an Architectural Perspective

Yongye Zhu, Shijia Wei, and Mohit Tiwari

Abstract—Web browser are traditionally benchmarked using user-centric page-load times. We found such metrics inaccurate from an architectural perspective, and limit hardware and systems optimization efforts, for performance and energy efficiency. In this article, we introduce *SysPLT*, an accurate and stable metric targeting system optimization for browsers. *SysPLT* tracks page load time based on browser's architectural activities. *SysPLT* captures on average 2.29× micro-architectural events compared to existing user-centric metrics, and provides insights to correct counter-intuitive system design choices recommended by user-centric metrics. We also show that *SysPLT* is more stable across repeated runs and different machine settings.

Index Terms—Browser, performance evaluation, micro-architecture

1 Introduction

Web browsers are among the most important applications running on computing devices today, with more than five billion users and more than ten billion device installations [1], [2]. They do not only run on mobile and PC devices. Remote browser services [3], [4], [5], render webpages on the cloud. They are commonly used in production for acceleration, privacy, system security and web development. *With the computational requirements of websites continuing to increase [6], optimizing system performance, including speed, energy efficiency, resource utilization, and quality of service, is crucial for running the browsers on systems ranging from battery-constrained devices to cloud servers.*

A time window of interest is needed for these interactive and long running browser applications to guide optimization efforts. The page load time is widely used for this purpose. For example, both industry and academic research use the page load time as the focus time window for browser workload characterization [7], [8], [9], [10] and optimization [11], [12], [13], [14], [15], [16]. These works aim to either reduce the overall page load time, or improve the efficiency and utilization within page load window.

Therefore, accurately tracking the browser page load time is fundamental to these optimizations. An inaccurate metric may mis-allocate research and engineering efforts. Existing metrics are driven by software and user perspectives. For example, the `loadEventEnd` is a software event that fires during the static HTML parsing sequence. It captures the entire object dependency graph during parsing. Other examples include user-perception metrics (e.g., First Contentful Paint or Largest Contentful Paint [17]). With a few (key) objects in the dependency graph tracked, they usually fire before the `loadEventEnd` event.

However, for optimizing overall system performance and efficiency, an architectural perspective is indispensable. Unfortunately, both the `loadEventEnd` event and user-perception timings lack such an architectural perspective. They capture only the *resource dependency* aspect, yet the dynamic computations and updates to the page objects are ignored. With increasingly complex and dynamic web design, dynamically computed and updated content contributes to a significant portion of the webpages. Yet, user-centric metrics conclude the page load time as soon as partial content is visible or interactive for the *first* time. Even the later `loadEventEnd` event fires before many dynamically fetched and updated objects are rendered in the browser. This dynamic behavior of the websites continues to drive computations, affecting battery life, device temperature, and interference to other applications, which are all part of end-to-end user experience.

In this paper, we introduce *SysPLT*, an architecture-focused page load time metric. The goal of *SysPLT* is to offer an end-to-end browser timing metric faithful to the underlying system and micro-architectural activities while being stable across repeated runs. Therefore *SysPLT* is designed specifically for systems research, while traditional page load metrics are better suited for directly user-centric optimizations. *The key observation underpinning* SysPLT *is that an architecture-faithful page load time needs to capture both the resource availability and the computational intensity of websites.* For example, if a website loads an image first and then performs computationally expensive 3D manipulations on it, an architecture-faithful page load time should include both the time needed to fetch the image and the compute time of the 3D effect. *SysPLT* achieves this by capturing both the activities in the network and in the micro-architecture.

Fig. 1 shows four 40-second micro-architectural traces of three popular real-world websites. Y-axis shows the number of micro-architectural events every 200ms in log-scale. Page load starts after a 5-second idle time. The purple line shows that the event `loadEventEnd` fires relatively early during the page load, ignoring a substantial amount of micro-architectural events afterwards. For only a few websites (e.g., the front-page of Amazon), this event is relatively faithful of the underlying system activities. For others, the `loadEventEnd` captures limited micro-architectural activities during page load. The orange dashed line shows that *SysPLT* captures on average $> 2.29\times$ micro-architectural events (Section 4). With *SysPLT* as a metric, performance and energy optimizations in hardware and browsers could be made more precise.

2 Background and Related Work

2.1 The Page Load Process

A browser loads a web page by first downloading the corresponding HTML, CSS and Javascript file from the network. The parsing of HTML and CSS files constructs the Document Object Model (DOM) and CSS Object Model (CSSOM). Each DOM node represents a resource object (e.g., videos and scripts) that will potentially be rendered on the screen. During the parsing of files, further resources are fetched and scripts are compiled and executed on demand. After the constructions, the DOM and CSSOM are combined with visual information to form a layout tree. Then the browser paints the page based on the layout tree. Deferred scripts may be executed after the first paint.

2.2 Browser and Website Performance Metrics

Page load time (PLT) metrics are widely used for benchmarking website and browser performance. `loadEventEnd` is one of the most well-known ones in the web community. It marks when the `load` event is fired by the browser. While `loadEventEnd` captures the resource dependency from the software perspective, it does not necessarily represent what users perceive and experience. This is because a browser would fire this event immediately with a

• *The authors are with the University of Texas at Austin, Austin, TX 78701 USA. E-mail: {yongye.zhu, shijiawei}@utexas.edu, tiwari@austin.utexas.edu.*

Manuscript received 15 July 2022; revised 11 September 2022; accepted 23 September 2022. Date of publication 29 September 2022; date of current version 27 October 2022.
This work was supported in part by Semiconductor Research Corporation (SRC) Task under Grant 2965.001, in part by Intel Side Channel Academic Programming (SCAP), and in part by Intel Resilient Architectures and Robust Electronics (RARE) research centers.
(Corresponding author: Mohit Tiwari.)
Digital Object Identifier no. 10.1109/LCA.2022.3210483

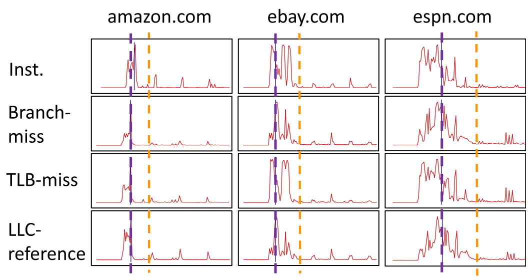

Fig. 1. Software only metric, `loadEventEnd` (purple line), only captures limited micro-architectural activities during page load. Our proposed metric *SysPLT* (orange dashed line) is able to capture $2.29\times$ more micro-architectural events (details in Section 4).

minimum page and defer fetching other content [17]. To tackle this problem, a set of metrics [18] targeting user experience (visibility and interactivity) was developed. For example, "First CPU Idle" measures the time a page takes to become minimally interactive.

3 *SysPLT* Design

Existing PLT metrics are not designed for system optimizations because they ignore system activities. In this section, we first identify three key requirements for a page load time metric that is faithful to the underlying system behavior:

- *R1*: An accurate architecture -centric PLT should mark the time when the browser enters an architectural idle state.
- *R2*: An accurate PLT should be the mark where most of the page resources (images, etc) are loaded into memory.
- *R3*: A useful PLT should be stable across repeated page loads in each distinct system setups.

SysPLT meets the above requirements by considering both resource availability and computational intensity during page loads, based on two key concepts: *Resource-idle* and *CPU-idle* .

The Resource-idle *point*: We observe that active page loading period has frequent resource requests, while idle phases have sporadic requests. Given the resource inter-arrival timings $InterRes[]$, we define the *Resource-idle* point T_r as the center of a monitoring window $[T_r - 1.0, T_r + 1.0]$ whose average resource inter-arrival timing is larger than or equal to a tail-percentile (e.g., 95) of those across all websites. Eq. (1) depicts this definition T_r

$$mean(InterRes[T_r - 1.0, T_r + 1.0]) > \\ = percentile(InterRes[ALLsites], 95). \quad (1)$$

The CPU-idle *Point*. Once the computation of the websites completes, the system will reach an idle or stable state. Our empirical characterization shows that most websites reach an idle or stable state after 30 seconds into the page load, with some websites exhibiting periodic yet infrequent activities. Therefore, given a dynamically committed instruction trace $Inst[]$, *CPU-idle* is defined as the center of a monitoring window whose average instruction count is less then or equal to a threshold% of the instruction count of the idle window of the same website. This idle window refers to time periods after the 30-second mark (e.g. $[30, 30+5]$). Eq. (2) formulizes this idle point T_c

$$mean(Inst[T_c - 1.0, T_c + 1.0]) < \\ = 75\% \times (Inst[30, 30+5]), \quad (2)$$

Putting the two together, *SysPLT* is defined as the first *CPU-idle* point after the first *Resource-idle*. By considering *CPU-idle*, *SysPLT* makes sure that the page load time is in an idle state, thus meeting requirement *R1*. With *Resource-idle*, *SysPLT* concludes the page load only after resource requests are fairly sparse, indicating most of the resources are already loaded (*R2*). With statistical criteria comparing the website loading behavior (Monitoring Window) to its own idle phase (Reference Window), profiling *SysPLT* is made application-independent.

Hyper-Parameterization. Note that in Eq. (1), the monitor window size is set to 2 seconds, the tail-percentile is 95%. In Eq. (2), the instruction count threshold is 75%, with the reference window starting at the 30th second and length set to 5 seconds.

However, to ensure a stable metric (*R3*), these parameters need to be systematically determined. We perform hyper-parameter selection under one single system configuration to determine a set of parameters that achieve better or similar page load time stability compared to `loadEventEnd`. These parameter are used when computing *SysPLT* for all other systems. We evaluate this set of parameters under five system configurations in Section 4.

TABLE 1
System Settings Used for Stability Evaluation

Resource	Browser	DVFS Governor	Symbol
Proxy □	Site Isolation Off △	Performance ⬠	□△⬠
Proxy □	Site Isolation Off △	On-demand ⬟	□△⬟
Proxy □	Site Isolation On ▲	On-demand ⬟	□▲⬟
Internet ■	Site Isolation Off △	On-demand ⬟	■△⬟
Internet ■	Site Isolation On ▲	On-demand ⬟	■▲⬟

4 Evaluation

SysPLT are evaluated using the Chrome browser (v85) with the Selenium [19] automation tool. We use 95 websites from Alexa-Top100 list (excluding 5 which failed with Selenium). We use hardware performance counters to collect micro-architectural events every 200ms for 40 seconds, and use the resource timing API in Chrome [20] for collecting network and resource activities. All experiments are performed on an Intel i7-6700K processor running Ubuntu 18.04 with 32GB DRAM and SMT enabled

$$\frac{1}{n}\sqrt{\sum_{i}^{n}\left(\frac{x_i}{\bar{x}} - 1\right)^2}. \quad (3)$$

In order to assess and improve the stability (*R3*) of *SysPLT* , we measure the Relative Standard Deviation (RSD, Eq. (3), also known as coefficient of variation) of the page load times across Alexa-Top100 websites. In Eq. (3), x_i is the collected metric for each run and $\bar{x}$ is the average. Lower RSDs indicate more stable the metrics. Results are compared with the `loadEventEnd` (LEE).

Table 1 lists five different system configurations used to evaluate *SysPLT* . The configurations vary CPU frequency, network latency, and browser concurrency settings. From row 1 to 5, the whole system becomes increasingly non-deterministic. With a network proxy, all resource requests are served from a proxy server on the local network instead of the Internet. When site isolation [21] is turned on, Chrome uses more than one distinct and concurrent processes for rendering each origin in a page. The `performance` governor in dynamic voltage and frequency scaling (DVFS) settings fixes the processor frequency to 4.0GHz, while `on-demand` dynamically scales the processor frequency between 800MHz and 4.0GHz depending on the running load.

4.1 Hyper-Parameter Selection

Hyper-parameter selection is performed under the most non-deterministic setting (row 5 in Table 1). Figs. 2, 3, and 4 plot the distribution of RSD (Eq. (3)) of all websites when varying reference window size, monitoring window size, and instruction count threshold, respectively. The distribution of RSD shows whether the metric provides a stable measurement of the page load process

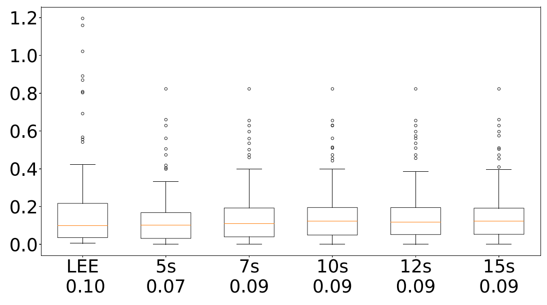

Fig. 2. *SysPLT* stability when varying *reference window size* in the most non-deterministic setting. X-axis shows `loadEventEnd` (LEE) and different reference window size from 5s to 15s. Geometric mean of RSD for all websites is annotated under the x-axis. SysPLT *outperforms LEE by reducing the number of outliers and increasing the stability of the most unstable websites.*

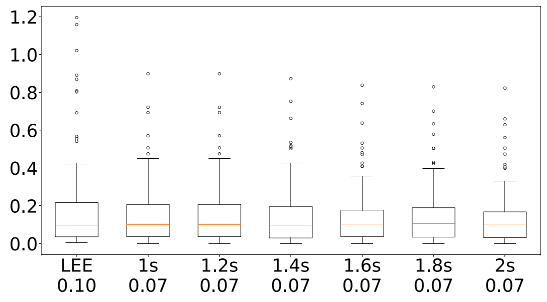

Fig. 3. *SysPLT* stability when varying *monitoring window size*. We fix the reference window to 5s and instruction count threshold to 75%. All of tested monitor window sizes have smaller max RSD and 2s provides the lowest third quartile and max among others.

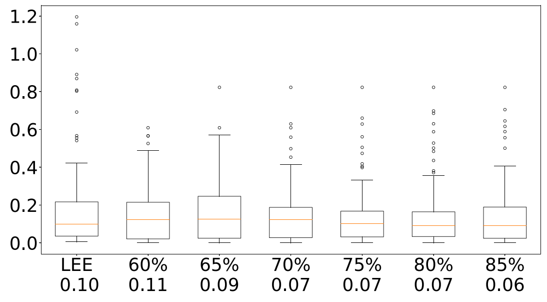

Fig. 4. *SysPLT* stability when varying *instruction count threshold*. The reference window is 5s and monitoring window is 2s. Although the 75% threshold shows a larger outlier compared to others, it has the lowest third quartile and maximum.

across all websites. The leftmost boxplots in all three figures show the RSD distribution of our baseline `loadEventEnd` (LEE) while the remaining boxplots plot RSD distribution of *SysPLT* with different hyper-parameter values. The geometric mean of RSD for all websites is annotated under the x-axis in each figure.

Fig. 2 shows the result of sweeping the reference window size from 5 to 15 seconds and fix the monitoring window at 2s and threshold at 75%. The LEE boxplot shows a wide distribution of RSD across all websites. While *SysPLT* , with its 0.07 overall geometric mean, (0.09 for LEE), outperforms `loadEventEnd` in terms of the overall stability and the outlier count in 95 websites. The 5s reference window size outperforms others in terms of geometric mean and third quartile. We perform sweeping on the monitoring window size from 1s to 2s and instruction count threshold from 60% to 85%. Figs. 3 and 4 compare the results. Similar experiments show that other hyper-parameters have little to negligible impact on the stability of *SysPLT* .

Our final configuration sets monitor window size to 2s and threshold to 75%, because in each sweep, they outperform others regarding lower quartile and geometric mean. While measured *SysPLT* will vary as we move to different microarchitectures or system configurations. However, the hyperparameters do not change. They are selected to achieve max stability in the most non-deterministic system settings.

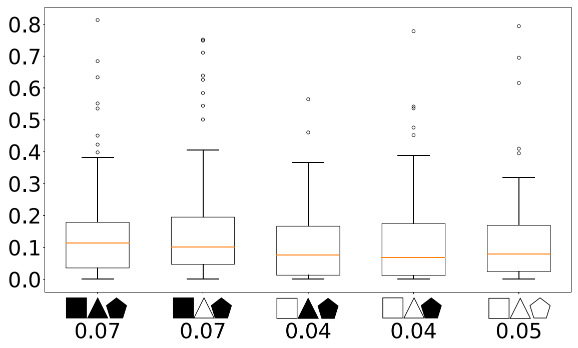

Fig. 5. *SysPLT* stability across different machine and network settings. Legends are defined in Table 1. When comparing settings with and without proxy (1st versus 3rd bar or 2nd versus 4th bar), the distribution of RSD narrows, showing that when networking becoming more stable, *SysPLT* becomes stable as well.

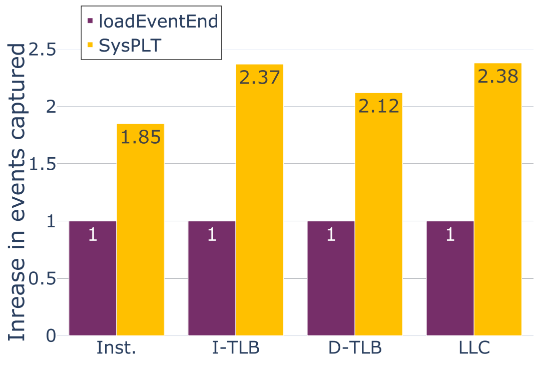

Fig. 6. Normalized improvements in number of micro-architectural activities captured with *SysPLT* .

4.2 Stability

With the selected hyper-parameters, we evaluate the stability of *SysPLT* in different software, hardware, and network settings (Table 1). Fig. 5 shows the RSD in 5 different settings. Comparing the 1st and 3rd bar from the left, we can see that when we eliminate the network non-determinism with a proxy, the geometric mean (value under the parameter) of RSD drops. This means that for a specific website, the *SysPLT* is more stable when enabling proxy. On the other hand, comparing site isolation on and off (3rd and 4th bar from the left), `on-demand` DFVS governor and `performance` DFVS governor (4th and 5th bar), the overall distribution as well as the geometric mean are lower than LEE. Lower value of RSD indicates a stable metric, meeting requirement *R3*.

4.3 Improvement on Counting Hardware Events

SysPLT measures on average $1.8\times$ longer page load time compared to `loadEventEnd`. Fig. 6 shows the normalized improvement on the number of micro-architectural events captured by *SysPLT* . Because *SysPLT* considers background activities, it helps to capture on average $2.37\times$ and $2.12\times$ more I-TLB and D-TLB events, in addition to $2.38\times$ Last-Level Cache accesses during the loading of 95 websites from AlexaTop100.

4.4 Case Study: Energy Efficiency Optimization

This case study showcases the benefits of *SysPLT* in guiding hardware design. In this experiment, we compare the total package energy, measured using Intel Running Average Power Limit (RAPL), between CPU running at 800MHz and 4GHz, during the page load using both LEE and *SysPLT* as the period of interest.

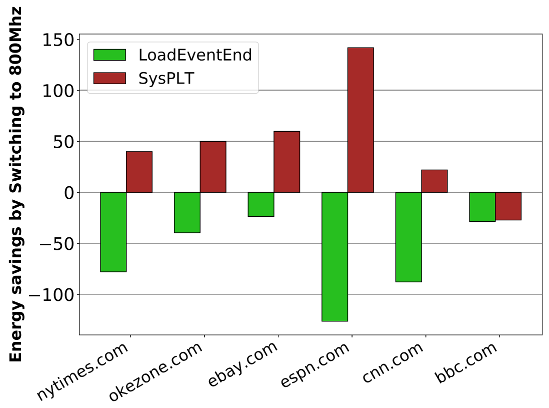

Fig. 7. Energy savings (J) when CPU switched to 800MHz (Negative values mean extra energy cost). LEE concludes that an 800MHz CPU consumes additional energy while *SysPLT* suggests that lower frequency offers better efficiency.

When evaluating a CPU with lower frequency, one may conclude, using LEE as the metric, that switching to 800MHz significantly increases the overall browser energy consumption as the LEE become $4-5\times$ longer. However, using *SysPLT* corrects this counter-intuitive conclusion by accurately mark when browser activities idle down. In fact, *SysPLT* shows energy savings in website loading when using a 800MHz CPU as depicted in Fig. 7.

5 Conclusion

This paper introduces *SysPLT* for *system optimizations* while traditional page load metrics are still recommended for user-centric optimizations. *SysPLT* considers both resource dependencies and underlying CPU/system activities to provide a stable, accurate and architecture-centric metric for the web page load process. With *SysPLT*, browser and processor designers can better focus their fine-grained analysis to optimize the modern web and systems.

Future Directions. While we demonstrated a case study on energy optimization using *SysPLT* , comprehensively investigating other benefits, e.g., guiding IPC improvements and extending *SysPLT* to heterogeneous systems with accelerators could promote wider adoption of *SysPLT* .

References

[1] Mozilla. Firefox public data report, 2022. Accessed: Jun. 13, 2022. [Online]. Available: https://data.firefox.com/dashboard/user-activity
[2] S. G. Stats, Browser market share worldwide, 2022. Accessed: Jun. 13, 2022. [Online]. Available: https://gs.statcounter.com/browser-market-share
[3] "Remote browser - co-browsing solution by remotehq," 2022. Accessed: Jul. 08, 2021. [Online]. Available: https://www.remotehq.com/remote-browser
[4] "Mighty | faster google chrome that uses 10x less memory," 2022. Accessed: Jul. 08, 2021. [Online]. Available: https://www.mightyapp.com/
[5] CloudFlare, "Protect teams with browser isolation | cloudflare," 2022. Accessed: Jul. 08, 2021. [Online]. Available: https://www.cloudflare.com/teams/browser-isolation/
[6] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Architect.*, 2013, pp. 13–24.
[7] A. Gutierrez et al., "Full-system analysis and characterization of interactive smartphone applications," in *Proc. IEEE Int. Symp. Workload Characterization*, 2011, pp. 81–90.
[8] D. Pandiyan, S.-Y. Lee, and C.-J. Wu, "Performance, energy characterizations and architectural implications of an emerging mobile platform benchmark suite-mobilebench," in *Proc. IEEE Int. Symp. Workload Characterization*, 2013, pp. 133–142.
[9] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with WProf," in *Proc. 10th USENIX Symp. Netw. Syst. Des. Implementation*, 2013, pp. 473–485.
[10] Y. Cao, J. Nejati, M. Wajahat, A. Balasubramanian, and A. Gandhi, "Deconstructing the energy consumption of the mobile page load," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 1, no. 1, pp. 1–25, 2017.
[11] Y. Zhu and V. J. Reddi, "High-performance and energy-efficient mobile web browsing on big/little systems," in *Proc. IEEE 19th Int. Symp. High Perform. Comput. Architect.*, 2013, pp. 13–24.
[12] Y. Zhu and V. J. Reddi, "WebCore: Architectural support for mobileweb browsing," *Proc. 31th Int. Symp. Comput. Architect.*, vol. 42, no. 3, pp. 541–552, 2014.
[13] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha, "Vroom: Accelerating the mobile web with server-aided dependency resolution," in *Proc. Conf. ACM Special Int. Group Data Commun.*, 2017, pp. 390–403.
[14] R. Netravali, A. Sivaraman, J. Mickens, and H. Balakrishnan, "Watchtower: Fast, secure mobile page loads using remote dependency resolution," in *Proc. 17th Annu. Int. Conf. Mobile Syst., Appl., Serv.*, 2019, pp. 430–443.
[15] R. Ko, J. Mickens, B. Loring, and R. Netravali, "Oblique: Accelerating page loads using symbolic execution," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 289–302.
[16] S. Mardani, M. Singh, and R. Netravali, "Fawkes: Faster mobile page loads via app-inspired static templating," in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, 2020, pp. 879–894.
[17] "User-centric performance metrics," Nov. 2019. Accessed: Oct. 11, 2021. [Online]. Available: https://web.dev/user-centric-performance-metrics/#important-metrics-to-measure
[18] "Performance audits," 2022. Accessed: Oct. 11, 2021. [Online]. Available: https://web.dev/lighthouse-performance/#metrics
[19] "Selenium," 2022. Accessed: Mar. 03, 2022. [Online]. Available: https://www.selenium.dev
[20] "loadEventEnd - MDN Web APIs," Sep. 2022. Accessed: Sep. 21, 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/PerformanceNavigationTiming/loadEventEnd
[21] C. Reise, "Mitigating Spectre with Site Isolation in Chrome," *Google Online Secur. Blog*, Jul. 2018. [Online]. Available: https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html