Fusing Speed Index during Web Page Loading

WEI LIU*, Tsinghua University, China
XINLEI YANG*, Tsinghua University, China
HAO LIN, Tsinghua University, China
ZHENHUA LI, Tsinghua University, China
FENG QIAN, University of Minnesota - Twin Cities, USA

With conventional web page load metrics (*e.g.,* Page Load Time) being blamed for deviating from actual user experiences, in recent years a more sensible and complex metric called Speed Index (SI) has been widely adopted to measure the user's quality of experience (QoE). In brief, SI indicates how quickly a page is filled up with above-the-fold visible elements (or *crucial elements* for short). To date, however, SI has been used as a metric for performance evaluation, rather than as an explicit heuristic to improve page loading. To demystify this, we examine the entire loading process of various pages and ascribe such incapability to three-fold fundamental uncertainties in terms of network, browser execution, and viewport size. In this paper, we design SipLoader, an SI-oriented page load scheduler through a novel *cumulative reactive scheduling* framework. It does not attempt to deal with uncertainties in advance or in one shot, but schedules page loading by "repairing" the anticipated (nearly) SI-optimal scheduling when uncertainties actually occur. This is achieved with a suite of efficient designs that fully exploit the cumulative nature of SI calculation. Evaluations show that SipLoader improves the median SI by 41%, and provides 1.43×–1.99× more benefits than state-of-the-art solutions.

CCS Concepts: • **Information systems → Browsers**; • **Networks → Network performance modeling**; • **Theory of computation → Scheduling algorithms**.

Additional Key Words and Phrases: speed index; web page loading; web performance; scheduling

ACM Reference Format:
Wei Liu, Xinlei Yang, Hao Lin, Zhenhua Li, and Feng Qian. 2022. Fusing Speed Index during Web Page Loading. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 23 (March 2022), 23 pages. https://doi.org/10.1145/3511214

1 INTRODUCTION

The ever-growing volume of web access has been stimulating the research and development of innovative techniques to enhance web page load performance [44, 48, 59]. To quantify the effect of existing techniques, simple and straightforward metrics like Page Load Time (PLT) [20], Time-to-First-Paint [25], Time-to-Largest-Paint [8], and Time-to-First-Byte [24] are the most used. Meanwhile, however, they have been blamed for deviating from actual user experience given the increasing complexity of today's web pages [40, 50, 60]. In response, several metrics that are more complex and sensible have been proposed in recent years, such as Speed Index [23], Above-the-Fold Time [33], Time-to-Interactivity [50], Byte Index [33], and Object Index [33]. Among them, we note

*Co-primary authors. Zhenhua Li is the corresponding author.

Authors' addresses: Wei Liu, Tsinghua University, Beijing, China, liuwei199803@gmail.com; Xinlei Yang, Tsinghua University, Beijing, China, yangxinlei19971105@gmail.com; Hao Lin, Tsinghua University, Beijing, China, linhao16@tsinghua.org.cn; Zhenhua Li, Tsinghua University, Beijing, China, lizhenhua1983@gmail.com; Feng Qian, University of Minnesota - Twin Cities, Minneapolis, Minnesota, USA, fengqian@umn.edu.

2476-1249/2022/3-ART23 $15.00
https://doi.org/10.1145/3511214

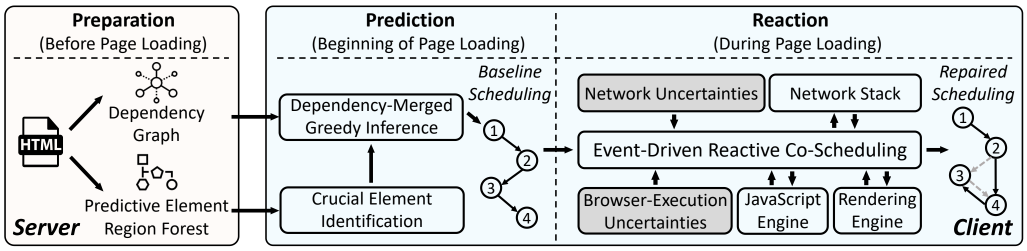


Fig. 1. Architectural overview of SipLoader, an SI-oriented web page loader that leverages the cumulative reactive scheduling framework.

that Speed Index (SI) has been widely adopted in both academia [45, 49, 57] and industry [23, 28, 38] to meticulously measure the user's quality of experience (QoE).

Different from traditional metrics that concentrate on a specific "milestone" event, SI tracks the entire page load process to quantify how quickly a page is filled up with above-the-fold visible elements (or *crucial elements* for short), thus closely reflecting the user's actual QoE. To be more specific, during a page's loading, one periodically samples a sequence of *visual completeness*, *i.e.,* the pixel distance between the current frame and the final frame of the fully-loaded page's above-the-fold area; then, SI is calculated (typically in an offline manner) as the integral of these visual completeness samples. Usually, a smaller SI indicates a better QoE [23].

To date, SI has been used to retrospectively measure the goodness of web page loading schemes, *e.g.,* Oblique [41], Fawkes [45], and Vroom [57], rather than as an explicit heuristic to actively direct page loading. This seems to be quite understandable considering the complicated, typically offline calculation process of SI [35, 47, 62]. Nevertheless, if there does exist a feasible approach to "fusing" SI during web page loading, we might be able to know the best SI one can achieve, or at least effectively improve SI by re-architecting today's web server and client design. In other words, our goal is to achieve or closely approach *SI-optimal page load scheduling* (or simply *SI-optimal scheduling*), which schedules the fetching and evaluation of web page objects (*e.g.,* images, stylesheets, and scripts) during page loading by explicitly and proactively taking SI into account.

If the scheduling variables are known beforehand or predictable, we find that SI-optimal scheduling can be determined ahead. In practice, however, these variables are subject to significant uncertainties that hinder the pathway to our goal. Uncertainties first stem from *network* and *browser execution.* Using one device to load the same static page, we notice remarkable SI difference under different network environments. In fact, even when this operation is repeated under very similar, controlled network environments, there is still considerable SI difference because the amount of computation resources allocated for loading the page can vary over time. Moreover, crucial elements are uncertain for the same page given the heterogeneous user-side *viewport sizes.* In particular, although the client knows the viewport size, it is unable to perform SI-optimal scheduling since it does not know which elements are crucial until the page is fully loaded. This is because many websites employ liquid layouts [1] to automatically adjust the sizes and locations of elements.

These uncertainties make it impossible to obtain SI-optimal scheduling in advance or in one shot without further adjustments. Hence, our key idea is to seek a feasible solution that is progressive and near-optimal. Through a comprehensive survey of literature, we seek out a promising framework from operational research, called *predictive-reactive scheduling* (or *reactive scheduling* for short) [37], which is dedicated to tackling a variety of scheduling uncertainties in a progressive manner, so as to minimize their negative impact and thus achieve near-optimal performance. Specifically, at the beginning of a page load, it generates a *predictive* SI-optimal (or says "baseline") scheduling; here predictive means that the scheduling is anticipated to be optimal if there were no upcoming uncertainties. After that, it reactively "repairs" the baseline scheduling when uncertainties (stemming from network, browser execution, *etc.*) actually occur to recover its (near-)optimality.

The key challenge of applying the above reactive scheduling framework to our goal is the lack of efficiency. Initially, we utilize the widely-adopted SI-calculation JavaScript library `Speedline` [3] and topological sort algorithm to implement a conventional approach to SI-optimal scheduling. Unfortunately, 94% page loading processes are then slowed down and sometimes even incur unresponsiveness of the web browser; in general, more complex pages that contain a larger number of crucial elements (and the related web page objects) experience severer decelerations.

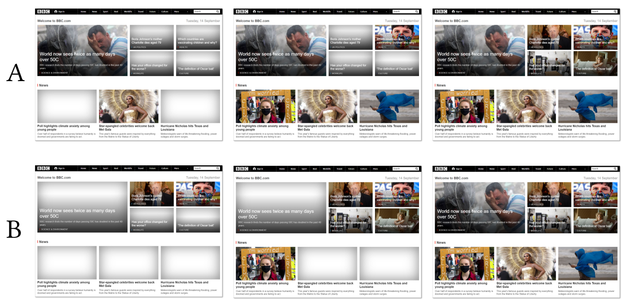
Motivated by the dilemma, we theoretically analyze SI-optimal scheduling problem and find the conventional approach to bear $O(n!)$ time complexity during the scheduling process, where n denotes the number of objects in a web page. In essence, the conventional approach treats the dependencies among web page objects as *constraints*, and thus utilizes topological sort to search for the SI-optimal scheduling in the entire constrained space. On the other hand, our key insight is that such dependencies are in fact beneficial to SI calculation given the cumulative nature of integral computation. Therefore, we devise the *dependency-merged greedy inference* algorithm, which strategically merges objects with their precedent dependencies to enable the usually suboptimal greedy local-search strategy to achieve near-optimal with a much lower time complexity of $O(n^2)$.

In addition to efficient SI-optimal scheduling when network and browser-execution uncertainties occur, we need to deal with the uncertainty of crucial elements caused by heterogeneous viewport sizes. Here we take a server-client collaborative approach for efficient identification of crucial elements. For a web page, the server first calculates the convex hull of each element's coverage regions under different viewport configurations offline, and then leverages computational geometry to cumulatively encode all the convex hulls into a *predictive element region forest*. When the client requests for the web page, it can quickly pinpoint the overlapping region of its viewport and all elements and thereby identify the crucial elements. While our implementation involves server-side collaboration to generate auxiliary data structures, the incurred overhead only includes $O(n \cdot log(n))$ computation complexity and typically <400 MB memory. Besides, the collaboration module is generic to all servers and dynamically loadable with simple configurations. We have developed it and released both its source code and executable binary.

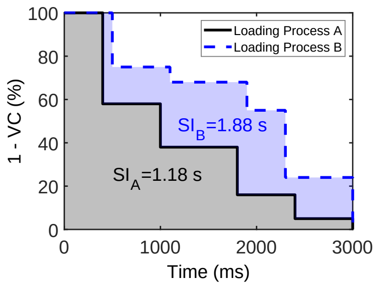
The novel framework that integrates our above efficient design is referred to as *cumulative reactive scheduling*, based on which we implement SipLoader, an SI-oriented page loader whose architecture is depicted in Figure 1. SipLoader first determines the baseline scheduling at the beginning of page loading by coordinating the server and client, and then carries out *event-driven reactive co-scheduling* at the client side to orchestrate network fetches, script executions, and element rendering, so as to achieve near-optimal scheduling in real time.

We conduct comprehensive evaluations for SipLoader on 300 sites with diverse client devices and network environments. Results show that for an average page, SipLoader can generate nearly SI-optimal scheduling in 12 ms, which is trivial (0.3%) compared to the page load time, with just 12 KB additional network traffic. Compared with state-of-the-art solutions like Vroom [57] and Fawkes [45], SipLoader provides 1.43×–1.99× more benefits with a median SI improvement of 41%.

Finally, despite our focus on SI in this work, we feel that the concept of fusing an evaluation criterion during web page loading is suited to other advanced or complicated metrics in the future. In particular, we have provided a suite of JavaScript interfaces to facilitate others' using the cumulative reactive scheduling framework. All the code and data in this work are publicly available at https://SipLoader.github.io/.



(a) 500 ms (b) 1200 ms (c) 3000 ms



(d) SI values of loading process A and B

Fig. 2. Comparing different loading processes of BBC.

2 BACKGROUND

To measure the performances of existing page load optimization techniques, a series of performance metrics are proposed, among which the most-used ones are typically simple and straightforward, such as Page Load Time (PLT) [20], Time to First Paint (TTFP) [25], Time to Largest Contentful Paint (LCP) [8], and Time to First Byte (TTFB) [24]. At the same time, however, these metrics have been receiving a growing concern that they cannot accurately reflect the actual user experience [34, 40], owing to their special focus on some specific "milestone" events. For example, PLT is defined as the trigger time of the JavaScript `onload` event, TTFP measures when the first pixel of the web page is painted onto the screen, and TTFB indicates the time when the client receives the first byte of the HTTP response. As a result, they all fail to comprehensively evaluate the entire page load process, especially for web pages with complicated contents.

Take two different page loading processes of the landing page of BBC [11] as an example in Figure 2. They share the same PLT (*i.e.,* both finish at 3,000 ms as shown in Figure 2c) while bringing totally different web browsing experiences to users. In particular, during the first loading process (Process A), large crucial elements are prioritized. Therefore, as shown in Figure 2b, the visible area of the page is quickly populated by large crucial elements. In contrast, during the second loading process (Process B), small crucial elements are loaded first. Consequently, the page is not as visually ready as that in Process A at 1200 ms (Figure 2b), thus resulting in worse QoE as users oftentimes expect web page contents to become visible quickly and progressively [2, 4, 5].

To highlight user experiences during page loading, several more complex and sensible metrics like Speed Index (SI) [23], Above-the-Fold Time (AFT) [32], Time-to-Interactivity (TTI) [50], Byte Index [33], and Object Index [33] are proposed in recent years. Among them, SI is the most popular one that has been widely adopted in both academia [45, 49, 57] and industry [23, 28, 38] to measure users' QoE in a fine-grained manner.

Specifically, SI judiciously quantifies how quickly a page is filled up with above-the-fold visible elements (*i.e.,* crucial elements) by tracking the entire page load process. It is formally defined as:

$$SI = \int_0^{AFT} (1 - VC(t))\, dt, \tag{1}$$

where $VC(t)$ denotes the visual completeness (*i.e.,* the pixel distance between the current rendered frame and the final frame of the fully-loaded page's above-the-fold area) at time t. In practice, one periodically (typically 100 ms) samples a sequence of visual completeness during a page load. Once the page is visually complete (*i.e.,* all the crucial elements have been loaded), SI is calculated as the integral of these visual completeness samples according to Equation 1. In the example of BBC,

Device	CPU	RAM	Network	Viewport	OS
PC-1	Intel i7-10700F (2.90 GHz)	64 GB	Residential broadband	2560 × 1440	Windows 10
PC-2	Intel i7-10700F (2.90 GHz)	64 GB	Residential broadband	1920 × 1080	Windows 10
PC-3	Intel E5-2420 (1.90 GHz)	32 GB	Residential broadband	1920 × 1080	Windows 10
Xiaomi XM11	Snapdragon 888 (2.84 GHz)	12 GB	LTE/5G	3200 × 1440	Android 11
Huawei HV30	Kirin 990 (2.86 GHz)	6 GB	LTE/5G	2400 × 1080	Android 10

Table 1. Client devices used for page loading tests.

Process A owns a lower SI value (1.18 s) than that of Process B (1.88 s) according to Figure 2d, which indicates that a smaller SI represents a better QoE.

Due to its meticulous characterization of the user's QoE during page loading, SI has been widely adopted by prior web accelerators, such as Oblique [41], Prophecy [49], WebGaze [40], Alohamora [39], WatchTower [52], Fawkes [45], and Vroom [57], to measure their effectiveness. Besides, SI has been utilized as a representative metric in popular web performance measurement tools like Lighthouse [17] and WebPageTest [28]. However, to date, SI has only been treated as a passive metric rather than an explicit heuristic to direct page loading. This is not strange since the calculation process of SI is rather complicated (as it involves integral calculation) and is typically conducted offline (as it requires the final frame of a page load as the input) [40]. Nevertheless, if there does exist a feasible approach to proactively employ SI to guide the page loading process, we would be able to make in-situ adjustments to the fetching and evaluation of web page objects during page loading so as to obtain or closely approach *SI-optimal page load scheduling*.

3 MOTIVATION

To explore whether there exists a practical approach to proactively exploiting SI as guidance to page load scheduling, as well as how this approach can be realized, we start with a comprehensive measurement study on the Alexa top 1,000 sites [9]. By comparing the pages' loading processes on different devices under diverse network environments (§3.1), we reveal that uncertainties stemming from network condition, browser execution, and user-side viewport size during page loading significantly hinder the pathway to our goal of achieving SI-optimal scheduling (§3.2).

3.1 Measurement Methodology

We carefully examine the entire loading process of the landing pages of the Alexa top 1,000 sites as of Jul. 16th, 2021. Every page is loaded online respectively on three PCs and two smartphones with different network environments, hardware configurations, and viewport sizes as listed in Table 1. In order to test page loading under diverse network conditions, for each PC, we throttle the bandwidth to {1, 10, 100, 1,000} Mbps (25 ms latency), and the latency to {10, 25, 50} ms (100 Mbps bandwidth) using Chrome DevTools [13]; for each smartphone, we load pages under real LTE and 5G networks, respectively.

We denote one *test group* as loading all the 1,000 pages 25 times, *i.e.*, 3 PCs × (4 bandwidths + 3 latencies) + 2 smartphones × 2 networks. In total, we have performed 30 groups of tests in one day, that is, 3 different time-of-day (0:00, 8:00, and 16:00) × 10 repetitions. All the page loading tests are performed on Chrome 91.0 with a cold cache setting. During each page load process, we also use Chrome DevTools to record fine-grained traces regarding CPU/memory usage, network transmission, web browser executions, and so forth for further analysis. Note that we also load pages on other browsers like Firefox, Safari, and Edge, but do not observe notable differences.

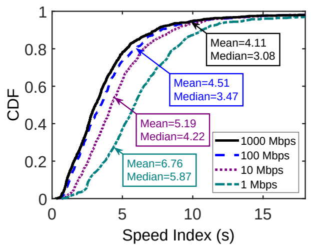


Fig. 3. SI distribution of the Alexa top 1,000 pages under different network bandwidths.

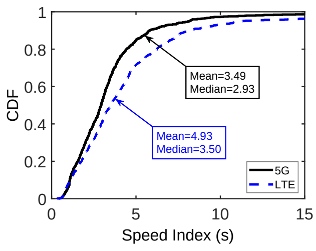


Fig. 4. SI distribution of the Alexa top 1,000 pages under LTE and 5G networks.

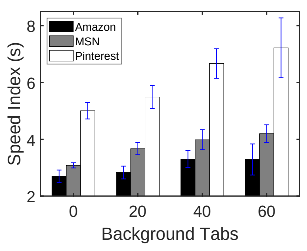


Fig. 5. SI of three representative pages with different numbers of background tabs opened.

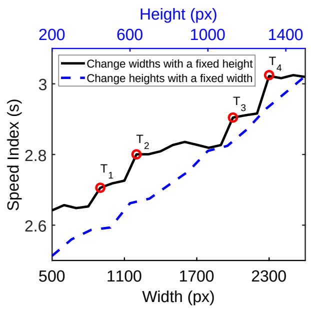
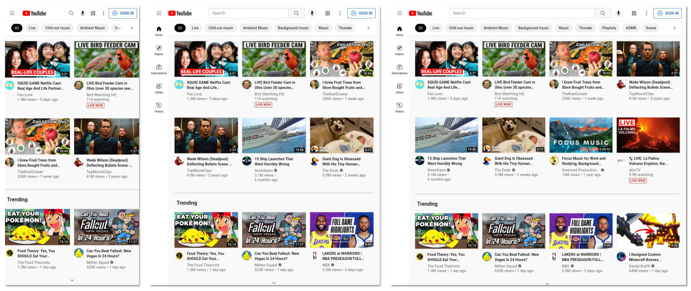


Fig. 6. SI of Youtube's landing page under different viewport widths and heights.



(a) 500 px (b) 872 px (c) 1128 px

Fig. 7. Liquid layouts of Youtube under different viewport widths.

3.2 Findings

In general, we observe three primary uncertainties deriving from network condition, browser execution, and client viewport size that significantly affect the SI of web page loading. As a result, they inevitably hinder us from achieving SI-optimal page load scheduling in one shot.

Network uncertainties. Network conditions affect the transmission of web objects during page loading, thus directly inducing disturbance on the SI of page loading. Figure 3 illustrates the cumulative distribution of SI values of the Alexa top 1,000 pages' loading processes on PC-3 under 1 Mbps, 10 Mbps, 100 Mbps, and 1,000 Mbps broadband networks. Similarly, Figure 4 shows the cumulative distribution of SI on Xiaomi XM11 under cellular networks including LTE and 5G networks. Each SI sample in Figure 3 and Figure 4 is calculated by averaging all the SI values of a certain web page's loading processes under the same network in different test groups.

From the above figures we observe that a higher bandwidth and a lower latency usually result in better SI values of page loading. In particular, page loads under the 100 Mbps network yield much lower SI values (4.51 s on average) than those under 1 Mbps (6.76 s on average) and 10 Mbps (5.19 s on average) networks. Page loads under the network with a 10-ms latency also have smaller SI values than those under the network with a 25-ms/50-ms latency (not shown). Meanwhile, the low-latency and high-bandwidth features of the 5G network also lead to better SI of page loading processes (3.49 s on average) compared with that under the LTE network (4.93 s on average). This is because the loading process is often bottlenecked by network transmission [55, 64], and thus accelerating the network can directly speed up the page loading, resulting in better SI.

On the other hand, as the network continues to improve, the SI values of page loading tend to converge to certain values. As depicted in Figure 3, when the bandwidth increases from 1 Mbps to 10 Mbps, the average SI values of page loading decrease significantly (from 6.76 s to 5.19 s); when the bandwidth further increases to 100 Mbps (4.51 s) and 1,000 Mbps (4.11 s), the decrease in SI continues to shrink. This is due to the fact that when the client-side network condition is good enough, the key bottleneck of page loading shifts from network transmission to client-side computation. Thus, further improving the network brings marginal benefits to SI.

Browser-execution uncertainties. Aside from network uncertainties, even when loading a same web page on a same device under similar network environments, we observe that the SI values of page loading may vary considerably over time. Given that page load performance is mainly determined by network conditions, client-side computation, and the web page itself [46, 47, 61], we attribute such SI variation to browser-execution uncertainties (*i.e.,* the computation resources allocated for loading the page are affected by other concurrent tasks processed on the client device), since the impacts of the other two factors on page load performance are minimized through our controlled, repeated experiments. Further, to quantify the effect caused by browser-execution uncertainties, we take preloaded background tabs (*i.e.,* browser tabs with web pages opened in the background) as concurrent tasks during page loading. Note that although not directly displayed on the screen, these background tabs also consume computation resources for event handling, server-client communications, and so forth, which can be monitored with Chrome DevTools.

In particular, we control the network environment of PC-3, and further load web pages on it with different numbers of background tabs. Figure 5 shows the SI values of loading three representative web pages including Amazon [10], MSN [18], and Pinterest [21] with {0, 20, 40, 60} background tabs opened. We clearly observe a linear increase and a growing variance in SI values as the number of background tabs increases. It indicates that a page's loading process can be slowed down or even becomes unpredictable if there exist plenty of concurrent tasks sharing local computation resources with it. In practice, the concurrent tasks on client devices during page loading are unknown beforehand, which makes browser-execution uncertainties a non-negligible obstacle that impedes the pre-computation of a page's SI-optimal scheduling.

Viewport size uncertainties. In addition to the network and browser execution, we notice that the size of the user-side viewport (*i.e.,* the region where web page contents can be viewed) could affect SI greatly. In particular, under controlled network environments and hardware configurations with no background tabs opened, the SI values of a same web page under different viewports differ (even if the traditional page load metrics like PLTs are nearly the same). Also, the SI of different web pages presents different trends with the change of user-side viewport sizes.

To better understand the phenomenon, we further perform finer-grained measurements, by loading web pages under viewports with a fixed height (1080 px) and widths ranging from 500 px to 2600 px, as well as viewports with a fixed width (1920 px) and heights ranging from 200 px to 1500 px. Intuitively, page loads under a larger viewport should have higher SI values, since a larger viewport usually displays more crucial elements to users, thus incurring additional overhead to the browser. However, Figure 6 shows the SI of YouTube's landing page under the above different viewports, and we can see that although SI increases when the viewport becomes wider, it presents a unique "staged" increase. For example, when the viewport width is between 500 px and 800 px, the SI values of page loads remain rather stable. When the viewport width increases from 800 px to 900 px, there is a sharp increase in SI (see point T_1 in Figure 6). Also, there exist another three "staged" SI increases when the viewport width increases from 1000 px to 2600 px (*i.e.,* T_2, T_3, and T_4 in Figure 6). In contrast, when the viewport height grows, SI increases smoothly.

Considering the actual appearance of YouTube's landing page, we ascribe such phenomenon to the *liquid layout* [1, 26] adopted in the page, which dynamically adjusts page's layouts according to the user viewport *width* [7, 26]. As shown in Figure 7, when viewport widths are set to 500 px, 872 px, and 1128 px respectively, the landing page of YouTube presents totally different layout schemes (*i.e.,* showing different numbers of image columns) with different numbers of crucial elements, which is the main reason for the sharp change in SI. Between these width thresholds, however, the page adjusts to the viewport through scaling, and the crucial elements remain largely unchanged, resulting in a stable SI. On the other side, when the viewport height increases, the landing page of YouTube never changes its layout scheme, and more originally below-the-fold elements become visible. Thus, SI linearly increases with the viewport height. For web pages using different liquid layout strategies, their viewport width thresholds that trigger the layout transition are unpredictable, which makes it even harder to perform SI-optimal scheduling on them.

In fact, other factors such as packet loss, browser cache status, and users' scaling also add up to the uncertainties. Nevertheless, they can be generally classified into the above three categories (*i.e.,* network, browser execution, and user-side viewport). These three types of uncertainties make obtaining SI-optimal scheduling in advance or in one shot impossible. Thus, we will next explore feasible approaches to practically achieving the goal.

4 DESIGN OF SIPLOADER

Our measurement study in §3 illustrates that uncertainties stemming from network, browser execution, and viewport size severely hinder us from achieving SI-optimal scheduling on web page object fetching and evaluation in advance or in one shot. Thus, we choose to obtain (nearly) SI-optimal scheduling in a progressive, reactive, and efficient manner. Our resulting solution is named SipLoader, an SI-oriented page load scheduler that realizes the philosophy by employing a novel *cumulative reactive scheduling* framework. As depicted in Figure 1, SipLoader implements the cumulative reactive scheduling framework with three core techniques.

- *Dependency-Merged Greedy Inference* calculates the baseline scheduling at the beginning of page loading on the client side. It is built based on the key insight that dependencies among web page objects are beneficial to SI calculation given the cumulative nature of integral computation. Therefore, it strategically merges nodes in the dependency graph with their precedent dependencies to enable the baseline scheduling to be calculated in a low complexity of $O(n^2)$.
- *Predictive Element Region Forest* efficiently handles the uncertainty of crucial elements incurred by heterogenous viewport sizes through a server-client collaborative approach. The server predicts each element's coverage regions under different viewport configurations, and then leverages computational geometry to cumulatively encode all the regions into a predictive element region forest. With this information, the client can quickly pinpoint the web page elements overlapped with its viewport (*i.e.,* crucial elements) during page loading.
- *Event-Driven Reactive Co-Scheduling* continues to repair the baseline scheduling when network and browser-execution uncertainties occur. It orchestrates network fetches and object evaluations in an event-driven manner, so as to achieve near-optimal scheduling in real time.

To be more specific, at the beginning of a page load, SipLoader efficiently identifies crucial elements based on predictive element region forest, and generates a predictive "baseline" scheduling for the web page (which is (nearly) SI-optimal without upcoming uncertainties considered) through dependency-merged greedy inference. Afterwards, it reactively "repairs" the baseline scheduling by performing event-driven reactive co-scheduling when uncertainties occur during the following page loading process, so as to recover the (near) optimality of the scheduling.

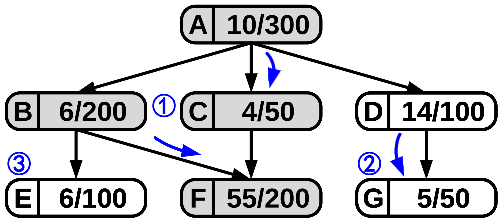
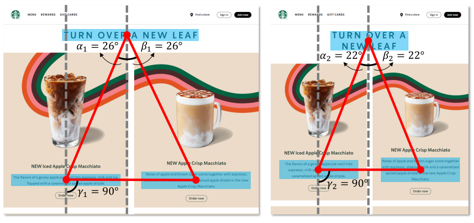


Fig. 8. An example of a dependency graph. Nodes in gray represent F's object group.



(a) 1400 px (b) 1080 px

Fig. 9. Relative angles between elements under different viewport widths of the same layout scheme.

4.1 Dependency-Merged Greedy Inference

The loading process of a web page should obey the dependencies among web page objects. For example, a script may read or write variables defined by other scripts; an element may also "read" CSS rules within style sheets to determine its presentation [48]. To ensure correctness, before loading an object, the browser must load its precedent dependencies first. Such dependency relationships among objects can be modeled as a directed acyclic graph, *i.e., dependency graph*.

Given a web page and its dependency graph, we then want to create the baseline scheduling for the web page. Also, we wish the baseline scheduling to be SI-optimal if there were no upcoming uncertainties. To achieve this, some apriori knowledge of the page is needed, including 1) SI gain of each object (*i.e.,* the visual contribution of each object to the above-the-fold page section) and 2) the time cost of loading each object. These two types of information are necessary for SI calculation.

To this end, we preload the web page on the server side and record fine-grained traces as described in §3.1, based on which the *expected* SI gain and the *expected* loading cost of each object can be estimated (discussed in §5). Figure 8 shows an example of the final dependency graph of a web page, where each node represents a web page object and each edge denotes the dependency between two objects. The SI gain (%)/loading cost (ms) of each object is attached to each node. With the dependency graph as well as objects' SI gain and loading cost, we can better formalize the problem of SI-optimal scheduling. Specifically, given a dependency graph where each node has an SI gain and a loading cost, we wonder in which order we should load web page objects such that the integration of unrendered fraction of a page's above-the-fold area over time is minimized (*i.e.,* minimize Equation 1), without violating the dependencies during page loading.

A conventional approach. A straightforward approach to calculating the above "anticipated" SI-optimal scheduling is to first generate all possible object loading orders that obey dependencies, and then select the one with the optimal SI value. We take advantage of a widely-adopted SI calculation JavaScript library, `Speedline` [3], together with the topological sort algorithm to implement a conventional approach to calculating the baseline scheduling. Here, the topological sort algorithm is used to generate feasible scheduling schemes that obey the dependencies among objects.

Since the calculation of SI-optimal scheduling depends much on the client-side viewport size (discussed in §4.2), we apply the conventional approach to the client. Specifically, at the beginning of page loading, the server sends the dependency graph along with the SI gain and loading cost information of each object to the client. The client then calculates the SI-optimal scheduling using the conventional approach. Unfortunately, we observe that 94% page loading processes are slowed down and sometimes pages even become unresponsive. Generally, more complex pages that contain a larger number of crucial elements suffer severer decelerations. We initially assume this is owing to the low execution efficiency of JavaScript [58]. Thus, we re-implement the conventional approach in WebAssembly [27], an emerging programming language that allows web applications to enjoy near-native runtime speed, but note that the situation does not change essentially.

To demystify the obstacle, we carefully dissect the conventional approach, and find that it bears a time complexity of $O(n!)$ where n denotes the number of web page objects to be scheduled. This is because the conventional approach seeks for all possible topological orders of object loading according to the dependencies, thus exploring the entire search space. Such high complexity of the conventional approach makes it hard to be practically applied to calculate the baseline scheduling.

Theoretical analysis. We then wonder whether there exists a more efficient solution to calculate the SI-optimal scheduling at the beginning of a page load. However, we find that the SI-optimal scheduling problem is NP-complete, which can be proved via a reduction from the **p**recedence **c**onstrained job **s**equencing problem on a single machine that minimizes total weighted completion time (or PCS problem for short) [30, 42].

Formally, the PCS problem is defined as follows. Given a directed acyclic graph $G = (N, E)$ with n nodes, each node $n_j \in N$ represents a job with a positive processing time p_j and a weight w_j. Job i must be completed before job j if there is a directed edge $e_{ij} \in E$ from node n_i to node n_j. On a single machine, it is obvious that for any given job execution sequence, the completion time T_j of job j is determined. The objective of the PCS problem is to minimize the weighted job completion time $\sum_{i=1}^{n} T_i w_i$, which has been proved to be NP-complete [42].

Now we prove that the SI-optimal scheduling problem is NP-complete through a reduction from the PCS problem. Our reduction takes any PCS instance $\langle G = (N, E), p, w \rangle$ and constructs the SI-optimal scheduling instance as follows. For each node $n_j \in N$, we define a web page object j, and set its loading cost c_j to be p_j, as well as its SI gain g_j to be $\frac{w_j}{\sum_{i=1}^{n} w_i}$. For each edge $e_{ij} \in E$, we define a dependency from object j to object i, *i.e.*, object i must be loaded before object j. It is clear that the construction can be carried out in polynomial time. For any given job execution sequence job s_1, job s_2, $\cdots$, job s_n (that obeys dependency constraints) in PCS problem, we have the corresponding object loading sequence object s_1, object s_2, $\cdots$, object s_n in SI-optimal scheduling problem. Then, recall the definition of SI according to Equation 1 and the calculation process shown in Figure 2d. The visual completeness when object s_j in the sequence is loaded is $1 - \sum_{i=1}^{j} g_{s_i}$, and we have:

$$
\begin{aligned}
SI = &\sum_{j=1}^{n} c_{s_j} \left(1 - \sum_{i=1}^{j-1} g_{s_i}\right) = \sum_{j=1}^{n} c_{s_j} \left(\sum_{i=j}^{n} g_{s_i}\right) \\
=&c_{s_1} \left(g_{s_1} + g_{s_2} + g_{s_3} + \cdots + g_{s_n}\right) + c_{s_2} \left(g_{s_2} + g_{s_3} + \cdots + g_{s_n}\right) + \cdots + c_{s_n} g_{s_n} \\
=&g_{s_1} c_{s_1} + g_{s_2} \left(c_{s_1} + c_{s_2}\right) + \cdots + g_{s_n} \left(c_{s_1} + c_{s_2} + c_{s_3} + \cdots + c_{s_n}\right) \\
=&\sum_{j=1}^{n} g_{s_j} \sum_{i=1}^{j} c_{s_i} = \sum_{j=1}^{n} g_{s_j} L_{s_j},
\end{aligned}
\tag{2}
$$

where $L_{s_j} = \sum_{i=1}^{j} c_{s_i}$ is the load completion time of object s_j. As the SI gain of loading a given object s_j in the sequence is $\frac{w_{s_j}}{\sum_{i=1}^{n} w_i}$, and the loading cost c_{s_j} of it is equivalent to p_{s_j}, we have:

$$
SI = \sum_{j=1}^{n} g_{s_j} L_{s_j} = \sum_{j=1}^{n} \frac{w_{s_j}}{\sum_{i=1}^{n} w_i} T_{s_j} = \left(\frac{1}{\sum_{i=1}^{n} w_i}\right) \sum_{j=1}^{n} w_{s_j} T_{s_j}. \tag{3}
$$

Note that the weight values w_j $(j = 1, 2, \cdots, n)$ are all positive. Thus, it is obvious that the job execution sequence in the original PCS problem achieves the minimal weighted completion time if and only if the corresponding object loading sequence in the SI-optimal scheduling problem is SI-minimal. Thus, the SI-optimal scheduling problem is NP-complete.

Our approach. Owing to the NP-completeness of the calculation of SI-optimal scheduling, we intend to seek a mechanism that efficiently generates a near-optimal scheduling as the baseline. We start with a very simple case. If there exists no dependency among different web page objects, it is obvious that the baseline scheduling should be a loading sequence where all the objects are sorted in an order according to their *SI gain efficiency* $e_j = \frac{g_j}{c_j}$ (or SI efficiency for short). Put simply, the higher SI efficiency value an object possesses, the earlier it should be loaded.

However, in practice the above scheduling scheme is often unachievable since it may violate the dependencies among objects. Fortunately, we observe that the cumulative nature of SI calculation greatly benefits from the dependencies, which facilitates us to efficiently generate a near-optimal scheduling. In particular, we define an object j together with all its precedent dependencies as j's *object group* (denoted as $G(j)$). For instance, in Figure 8, object F's object group is marked in gray. If a browser tends to load F, it has to first load all the other objects in F's object group, *i.e.,* F's precedent dependencies. Therefore, when taking dependencies among different objects into consideration, the SI efficiency of a certain object j turns out to be an overall SI efficiency of loading j's object group, which can be formally defined as the *merged* SI gain efficiency (or merged efficiency for short) of an object j: $\bar{e}_j = \frac{\sum_{i \in G(j)} g_i}{\sum_{i \in G(j)} c_i}$.

SipLoader assumes that to achieve a (near-)optimal scheduling, the browser should continue to load objects from the object group with the highest merged efficiency (termed as optimal object group) since these objects own the highest SI gain expectations. Guided by this, we devise the dependency-merged greedy inference mechanism that keeps detecting and loading objects from the optimal object groups, so as to efficiently generate a near-optimal scheduling.

Specifically, SipLoader first calculates the merged efficiency of each object on the entire dependency graph and then selects the optimal object group (denoted as object k's object group) as it owns the highest SI gain expectation. Before loading object k, SipLoader needs to load k's precedent dependencies. Therefore, SipLoader further searches in k's precedent dependencies (which form a subgraph of the original dependency graph) to find the optimal object group among them. Such operation is repeatedly performed until there is only one object left, which is selected to be loaded next. Note that loading this selected object will not violate the dependencies since after the above processes, only the object with no precedent dependencies can be left. The corresponding node of this object is then deleted from the original dependency graph. Afterwards, SipLoader updates the merged efficiencies of the remaining objects for iteratively inferring the next object to be loaded.

We go through the iteration processes of our greedy algorithm on the example dependency graph in Figure 8. Initially, we calculate the merged efficiency $\bar{e}$ for each object, and we find object F has higher a merged efficiency $\bar{e}_F = \frac{g_A+g_B+g_C+g_F}{c_A+c_B+c_C+c_F} = 0.1$ than any other object. Thus, we then search in F's precedent dependencies, *i.e.,* A, B, and C, for the first object to load. Among objects A, B, and C, C has the highest merged efficiency $\bar{e}_C = \frac{g_A+g_C}{c_A+c_C} = 0.04$, and we search in its precedent dependencies, wherein only object A is left. As a result, A is selected to be loaded first. In fact, A must be loaded first since it is currently the only object in the dependency graph that has no precedent dependencies. Then, we delete A from the dependency graph and update the merged efficiency for the remaining objects. In the next iteration, F still has the highest merged efficiency, and we search in its precedent dependencies, *i.e.,* B, and C, and choose C as the second object to be loaded. Similarly, objects B and F will be chosen as the third and the fourth object to be loaded respectively (①), with objects D, E, and G remaining. Given that D has the highest merged efficiency at present ($\bar{e}_D = 0.14$) without precedent dependencies, it is selected as the fifth object and removed from the graph. Finally, G has a higher merged efficiency ($\bar{e}_G = 0.1$) than E ($\bar{e}_E = 0.06$), and thus G will be loaded next (②) and E will be loaded in the end (③).

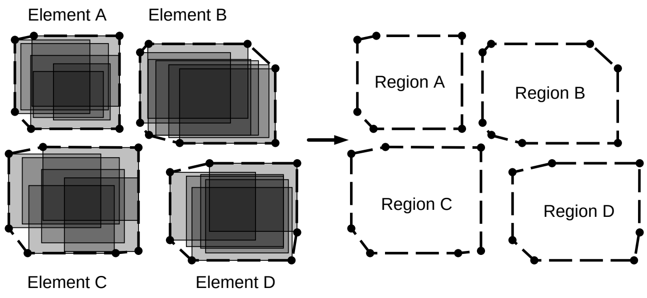


Fig. 10. Predictive element regions of four example elements.

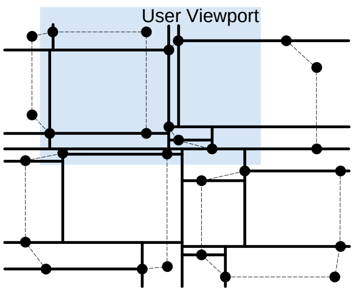


Fig. 11. Querying crucial elements at the client side with the k-dimensional tree.

Employing the above mechanism, we can generate a (near-)optimal scheduling in an $O(n^2)$ complexity, where n represents the number of nodes in the dependency graph. We will further discuss the efficiency and optimality of our approach in §6.

4.2 Predictive Element Region Forest

In order to generate the baseline scheduling with dependency-merged greedy inference at the beginning of page loading, we need to determine crucial elements in advance so that we can assign SI gains to the corresponding web page objects. Nevertheless, neither the server nor the client can achieve this on its own in practice. On the one hand, the client cannot determine the crucial elements until the page is fully loaded, since the liquid layout schemes adopted in many pages can make elements' sizes and positions elusive on different devices as mentioned in §3.2. On the other side, the server alone cannot determine the crucial elements for the client as well, since it does not know the client's viewport size beforehand. Although the client can request crucial element information from the server by sending the viewport size at the beginning of page loading, it will incur additional latency and decrease page load efficiency. This is because the browser must first receive and execute the scripts that explicitly define such logic (which at least needs a round-trip time), and then suspend page loading until the server returns the crucial element information.

To tackle this dilemma, we leverage a client-server collaborative crucial element identification approach based on our two-fold observations. First, almost all web pages' layout structures are only affected by the width of a client viewport (while the height changes are usually handled by page scrolling). Second, although an element's absolute position varies with the viewport width, its relative position to other elements can remain largely unchanged within the same layout scheme. For example, Figure 9 shows the same web page under different viewport widths, where elements' sizes and absolute positions under screen coordinates are apparently different. However, the relative positions between elements, *e.g.*, the *relative angles* $\{\alpha_1, \beta_1, \gamma_1\}$ and $\{\alpha_2, \beta_2, \gamma_2\}$ that characterize the inclination of lines between geometric centers of two elements, are extremely similar. Note that under different layout schemes, the relative positions between elements can vary greatly, *e.g.*, in Figure 7, some elements shift from the second row to the first row after the layout transition.

Inspired by the above, SipLoader determines the crucial elements in the following way. First, the server repeatedly preloads the target web page under a series of viewport widths (from 500 px to 2500 px every 50 pixels) and records the possible sizes, absolute positions, and relative positions for each element. For two page loads, when their elements' relative positions are similar, *i.e.*, the average distinction of relative angles is less than a certain threshold δ, they are classified into the same layout scheme. Currently, we set δ as 5°, which is found to balance well the tradeoff between the accuracy of the classification and the scale of identified layout schemes. It is worth noting that a too large δ leads to many false positives in classification; while an excessively small δ may incur an explosion in the scale of layout schemes, imposing a heavy burden on further calculations and data transmissions between clients and servers.

For each identified layout scheme of the page, we cumulatively overlay the coverage regions of each element under different viewport widths. Figure 10 shows the overlaying process of four elements' coverage regions, where each gray rectangle is the bounding rectangle of the element [15] under a specific viewport width. The overlaid coverage region of an element (termed as predictive element region) predicts possible positions where the element may appear under the given layout scheme. We then leverage the Graham scan algorithm [31] to calculate the convex hull for each predictive element region so as to encode region data. Also, we record the area of each region for determining the SI gain of the related objects (§5). The convex hulls of all predictive element regions of a layout scheme are constructed into a k-dimensional tree [56], which will be assembled together with other different layout schemes' (termed as predictive element region forest).

At the beginning of page loading, the predictive element region forest will be sent to the client. The client chooses a predicted layout scheme from the forest based on its actual viewport size, and then infer the crucial elements by querying the corresponding k-dimensional tree (as shown in Figure 11). For web pages that have tens of thousands of elements, the $O(\sqrt{n})$ complexity of k-dimensional tree query in JavaScript can have negative impacts on page load performance. To cope with this, once a vertex of a certain convex hull is found to be overlapping with the viewport during the query, we determine the corresponding element to be a crucial element, and then prune the tree by removing its convex hull's remaining vertexes, so as to reduce the search space.

With the help of the above efficient computational geometry algorithms, the time complexity of the server-side preprocessing is only $O(n \cdot log(n))$ [53], where n denotes the number of web page elements. In addition, the collaboration module is generic to all servers and dynamically loadable with simple configurations.

4.3 Event-Driven Reactive Co-Scheduling

Although SipLoader manages to calculate the baseline scheduling through techniques discussed in §4.1 and §4.2, such a scheduling scheme is often suboptimal in practice due to uncertainties that lie in network transmission and browser executions. For instance, when a web page is preloaded on the server where object A and B share a same SI gain but A induces a lower loading cost than B, A will be placed ahead of B in the baseline scheduling. However, it is possible that when the page is loaded on a client, B arrives much earlier than A due to network uncertainties. In this case, inflexibly obeying the baseline scheduling if B has no precedent dependencies (*i.e.,* pending object B and waiting for A) could lead to a suboptimal scheduling result. In addition, as mentioned in §3, the background activities of other concurrently opened tabs also add to the uncertainties of browser evaluation time on each object, which can result in a suboptimal scheduling as well.

To cope with the above issues, SipLoader adopts an event-driven reactive co-scheduling scheme that dynamically repairs the baseline scheduling to achieve a near-optimal page loading scheme in real time. In particular, SipLoader orchestrates network fetches, script executions, and element rendering by maintaining a sending queue and a receiving pool. The sending queue holds network requests that are currently not ready to be sent, *e.g.,* those blocked by browser's maximal concurrent connection constraints [36]. The receiving pool holds objects that have been fetched to the client but are not ready for evaluation yet, *e.g.,* those whose precedent dependencies are not fully loaded.

As shown in Figure 12, SipLoader first initializes the sending queue with all the object requests sorted according to the baseline scheduling. The requests ranking ahead are popped up and handed over to the network stack for concurrent fetching (the number of concurrent requests is determined

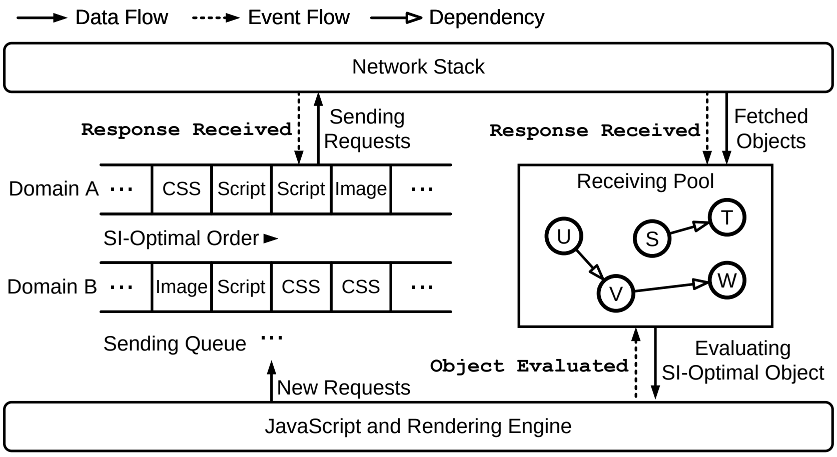


Fig. 12. Architectural overview of event-driven reactive co-scheduling.

by the maximal network concurrency supported by the browser), while the others are held in the sending queue. Then, SipLoader keeps monitoring the events generated from the network stack and browser engines. Once a response is received, *i.e.,* a `Response Received` event is generated by the network stack, the fetched object is added into the receiving pool. Meanwhile, SipLoader pops up the first request in the sending queue and passes it to the network stack. On the other side, when an object's evaluation is completed, *i.e.,* an `Object Evaluated` event comes up, SipLoader selects an object from the receiving pool which is headmost in the baseline scheduling and has no unloaded precedent dependencies. This object is delivered to the JavaScript or rendering engine for evaluation. In this way, SipLoader adaptively co-schedules network fetches and local executions so as to achieve the near-optimal scheduling in real time.

5 IMPLEMENTATION

The implementation of SipLoader mainly focuses on scheduling images, stylesheets, and JavaScript files, since they account for 81% of today's web objects in total [43], but note that other types of web page objects like fonts can be accommodated in a similar way.

Dependency graph generation. We capture the fine-grained dependency graph of a web page by preloading the page at the server side and monitoring the entire page load process with Chrome DevTools Protocol [14]. To estimate SI gain for each object, we first identify the web page elements that an object can have *visual impacts* on. Specifically, for image objects, their impacted web page elements are the image tags holding them. For stylesheets, their impacted elements are those affected by their CSS rules. In terms of scripts, we use JavaScript `MutationObserver` API [19] to detect the changed elements during the script execution, and we regard the visually changed elements as their impacted elements. Based on this, we estimate the SI gain for each object at the beginning of page loading, by summing up all the areas of its impacted crucial elements' predictive element regions, and then divide the result by the viewport area.

Next, we discuss how we estimate the loading cost for each object based on the page load traces collected on the server side. The estimated loading cost of an object consists of two parts, *i.e.,* network transmission time and browser evaluation time. Due to the parallel nature of network requests during page loading, we divide the time into 10-ms slices to better perform the estimation. Specifically, an object's network transmission time is calculated by summing up the average network transmission time that they share with other objects in each slice. For example, if there are four concurrent requests in a same 10-ms slice, the average network transmission time of each object within this time slice is 2.5 ms. Adding up all the averaged transmission time in each slice, we get the estimated network transmission time for each object.

In terms of browser evaluation time, for script files, we use their actual execution time recorded in browser traces during page loading. For stylesheets, since today's browsers do not provide their fine-grained timing information in traces, we calculate their browser evaluation time as the sum of the `RecalculateStyle [6]` time of an average CSS rule within them. Similarly, we set the browser evaluation time of each image object to be the rendering time of an average image. We realize all the above using Python and Node.js together with Puppeteer libraries [22].

Web page rewriting. In order to incorporate SipLoader into page loads on top of unmodified commodity browsers, we rewrite the web page on the server side using Beautiful Soup [12] with Python. We delete all image, CSS, and script URLs defined in the HTML file so that we can schedule the fetches and evaluations of the three types of objects. Then, we pack the modified HTML file, predictive element region forest, dependency graph, and the SipLoader codes (in JavaScript) into a new HTML file. When the client requests for the page, the server will send the rewritten version of the HTML file instead. The client loads the rewritten page as common web pages, but during the loading process, the SipLoader codes control both network fetches and object evaluations.

6 EVALUATION

Our evaluation of SipLoader consists of two parts. First, we demonstrate that SipLoader significantly improves the median SI by 41%, and provides 1.43×–1.99× more benefits than state-of-the-art solutions (§6.2). Then, we evaluate the benefits of SipLoader's each component (§6.3).

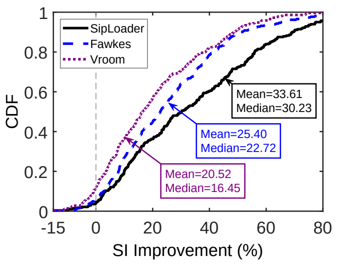
6.1 Experiment Setup

We compare SipLoader with two state-of-the-art web page accelerators, Vroom [57] and Fawkes [45]. Vroom improves page load speed by scheduling resource fetching during page loading based on a server-aided dependency resolution scheme. In terms of Fawkes, it accelerates web page loads by generating app-like static HTML templates for web pages offline, and patches dynamic contents when pages are being loaded. We choose these two web page accelerators since the former adopts a dependency-driven object fetch scheduling, and the latter represents a typical visibility-aware page load scheme, both of which have commonalities with the design of SipLoader.

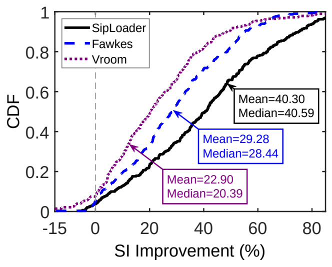
Our evaluations mainly comprise the landing pages of 300 websites randomly chosen from the Alexa top 1,000 sites, covering the categories of news, online shopping, sports, business, *etc.* To create a reproducible test environment as well as enable the page rewriting procedures involved in all the three accelerators, we use Mahimahi [51], a web record-and-replay tool, to record the corpus pages. Web pages rewritten by the three accelerators are all deployed on top of a local server (Ubuntu 20.04, 2.9 GHz × 16 core CPU, 16-GB memory, and 1-Gbps NIC).

We conduct evaluations on both mobile and PC devices as previously listed in Table 1. Although Vroom and Fawkes primarily focus on mobile environments where client computation is usually the page load bottleneck, their core designs are compatible with desktops. Also, we feel desktop clients may still benefit from them since as we mentioned in §3.2, both network and client computation are often uncertain during page loading. To perform apple-to-apple comparisons under the same test environment, we re-implement Vroom and Fawkes based on their key mechanisms[1]. All PC devices connect to the local server via an in-lab LAN with a bandwidth of 1 Gbps for controlled

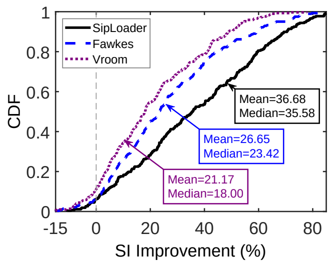
[1]Owing to a small set of devices we use, we did not implement the components of Vroom that are related to device-specific customization for simplicity.



(a) 100 Mbps bandwidth, 25 ms latency, 2.40 GHz CPU

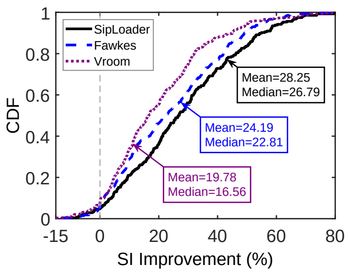


(b) 10 Mbps bandwidth, 50 ms latency, 2.40 GHz CPU

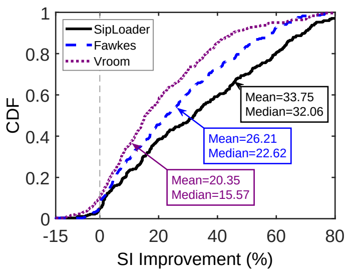


(c) 100 Mbps bandwidth, 25 ms latency, 1.90 GHz CPU

Fig. 13. SI improvements under desktop, cold cache settings.



(a) 5G



(b) LTE

Fig. 14. SI improvements under mobile, cold cache settings.

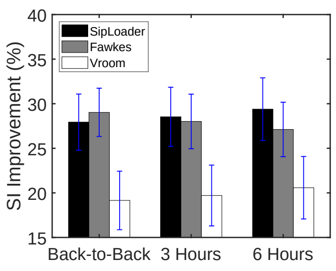


Fig. 15. SI improvements under 100 Mbps bandwidth, 25 ms latency, and warm cache settings.

experiments ({1, 10, 100} Mbps bandwidth and {10, 25, 50, 75} ms latency), while two smartphones connect to the local server via cellular networks including LTE and 5G networks. It is worth noting that such experiment setup has covered the three major uncertainties (*i.e.,* network, client-side computation, and viewport sizes) that can affect the optimality of scheduling during page loading.

We evaluate the performance of three accelerators in both cold cache and warm cache ({0, 3, 6} hours) settings. In particular, for Fawkes' cold cache settings, we generate its HTML templates 24 hours apart. Under each environment configuration, every page is loaded on Chrome 91.0 with 5 repetitions, and we use the average SI value of the 5 loads of a page as the final results. We leverage Chrome DevTools to record the visual progresses of web page loading, which are used for subsequent SI calculation with `Speedline`.

6.2 End-to-End Performance

Cold cache. Figure 13a shows the SI improvements of three web page accelerators on PC-2 under the 100-Mbps broadband network with a 25-ms latency. In such a cold cache setting, the median and average SI improvements of SipLoader are 30.23% and 33.61% respectively, significantly outperforming the other two web page accelerators. In particular, the median SI improvement with SipLoader is 1.84 × higher than Vroom's, and 1.33 × higher than Fawkes'. This is because even if Vroom prioritizes the fetches of web page objects that are in the critical path of page loading, it does not consider objects' visual impacts, yielding low improvements on SI. Moreover, although Fawkes' immediate rendering of static HTML templates speeds up the visual progress during page loading, the benefits are only restricted to the beginning of a page load. After the static parts of a web page are rendered, Fawkes' loading process of dynamic parts does not account for the visual importance of web page objects as well.

When network conditions become worse, SipLoader's benefits increase. As shown in Figure 13b, when the network condition degrades from 100 Mbps with a 25-ms latency to 10 Mbps with a 50-ms latency, the median and average SI improvements of SipLoader increase to 40.59% and 40.30%, respectively. Compared with Vroom and Fawkes, SipLoader enjoys larger SI improvements as network conditions get worse. The average SI benefits of SipLoader increase by 6.69% after the degradation, while those of Fawkes and Vroom are only 3.88% and 2.38%. This is because SipLoader fetches and evaluates web page objects based on their *merged SI gain efficiencies*. Poor network conditions require a more efficient utilization of available network capacities to fetch objects with higher SI gains and lower transmission time. Similarly, when client performances are relatively poor, SipLoader provides the highest SI improvements among three accelerators (as shown in Figure 13c), as it gives priority to evaluating objects with higher SI gains and lower evaluation time.

Figure 14 shows the SI improvements of the tested accelerators under 5G and LTE mobile environments. SipLoader keeps outperforming Vroom and Fakes in both scenarios. Also, the average benefits of SipLoader on the LTE network are more significant than those on the 5G network (33.75% *v.s.* 28.25%), owing to the higher latency and lower bandwidth of the LTE network. We omit the results of other environment configurations as their trends are basically the same.

Warm cache. We also evaluate SipLoader in warm cache settings. As shown in Figure 15, for back-to-back loads (*i.e.,* pages are loaded immediately after preceding warming-up loads), Fawkes presents a higher improvement on average SI over the corpus pages than SipLoader and Vroom. In detail, the average improvement on SI of Fawkes reaches 29.02%, while that of SipLoader and Vroom are 27.93% and 19.15%, respectively. We attribute this to Fawkes' static HTML templating feature that caches unchanged objects across different versions of web pages. When pages are loaded in a back-to-back manner, the vast majority of objects within the pages remain unchanged [45], thus incurring little overhead of Fawkes' dynamic content patching and bringing significant SI improvements. Despite SipLoader's fewer SI benefits than Fawkes', the differences are subtle. This is due to the fact that when objects can be quickly fetched from the browser cache (*i.e.,* network is no longer one of the bottlenecks during page loading), SipLoader keeps evaluating them in a (nearly) SI-optimal order, and therefore is still able to improve SI. On the other side, after prioritizing resource fetches and receiving prioritized objects from the browser cache, Vroom does not further schedule the object evaluation based on any visibility-aware heuristics. Thus, its benefits on improving SI are rather limited compared with SipLoader and Fawkes.

In terms of 3-hour and 6-hour warm cache settings, SipLoader continues to outperform the other accelerators because, as time goes by, the number of changed objects within web pages increases, and thus more objects need to be transmitted through the network, which makes scheduling the object fetching process more important to improving SI. In contrast, the benefits of Fawkes gradually drop since more objects are loaded through dynamic patching rather than from the cached HTML templates.

Sensitivity analysis. Although SipLoader presents promising improvements on the SI of page loads in most scenarios, we do notice performance degradation (5%–15%) in some web pages. Delving deeper, we find that these pages are quite simple and have very few page objects (typically no more than 20 objects). This is reasonable since the crucial element identification and the scheduling process of SipLoader require extra time, which cannot be compensated by SipLoader's benefits on these pages. Thus, for pages with a small number of objects (*i.e.,* ≤20 objects), we can simply leave them unmodified to avoid performance degradation.

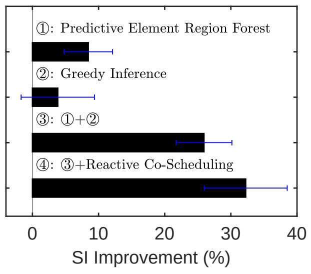


Fig. 16. Impacts of individual modules of SipLoader on the 100 Mbps network.

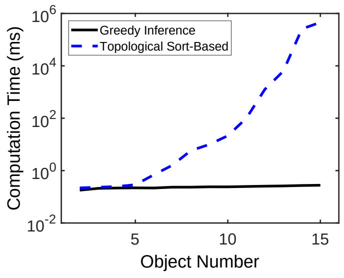


Fig. 17. Comparing the efficiency of two approaches to calculating the (nearly) SI-optimal scheduling.

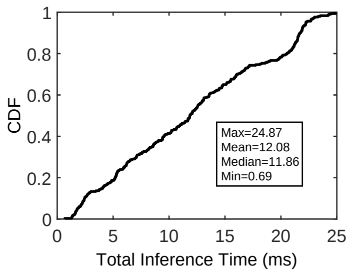


Fig. 18. Time consumption of inferring baseline scheduling.

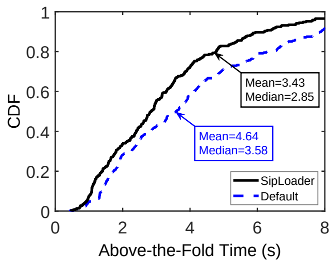


Fig. 19. Above-the-Fold Time with and without SipLoader.

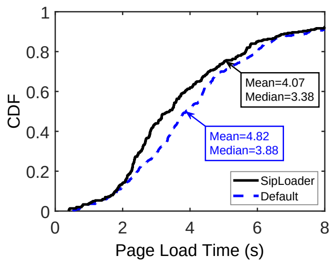


Fig. 20. Page Load Time with and without SipLoader.

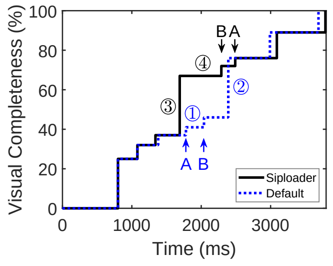


Fig. 21. Visual progress with and without SipLoader.

6.3 Individual Components

To better understand the benefits of SipLoader, we evaluate its individual components to uncover their impacts by incrementally enabling one at a time. When the mechanism of predictive element region forest is disabled, we regard all of the web page elements as crucial elements. When dependency-merged greedy inference is disabled, we use the loading order of the default web page as the substitute. When event-driven reactive co-scheduling is disabled, we fetch and evaluate objects strictly according to the baseline scheduling without repairing it. We perform the evaluation on top of PC-3 with our 300 corpus pages under the 100 Mbps network with cold cache settings.

Predictive element region forest. After the client determines crucial elements in the current user viewport based on the predictive element region forest sent from the server, we dynamically append `link` tags to the header of the DOM tree, with their `rel` attributes set as "preload" and their `href` attributes set as the URLs of objects related to crucial elements. In this way of utilizing *HTTP/2 Preload* policy, the browser will fetch objects that have visual impacts on crucial elements in advance for future evaluations. As shown in Figure 16 (①), when only the mechanism of predictive element region forest is enabled, we enjoy an average SI improvement of 8.45%, indicating that prioritizing the fetches of objects related to crucial elements is important to SI improvements.

Dependency-merged greedy inference. We then only enable dependency-merged greedy inference on SipLoader, and attempt to infer the (nearly) SI-optimal page load scheduling over all of the web page objects (*i.e.,* all web page elements are regarded as crucial elements). As shown in Figure 16 (②), this brings few SI improvements (3.83% on average). The main reason is that inferring the baseline scheduling over all web page objects can hardly obtain the (nearly) SI-optimal loading orders, since not all objects have visual impacts on users. For example, some web pages may hold a large portion of contents in their below-the-fold sections, which have a higher probability to be prioritized when only the greedy inference is used. These contents do not contribute to the visual completeness during page loading, and worse still, they can delay the rendering of above-the-fold visible page contents, resulting in a high SI value.

The unsatisfactory performance of the single use of dependency-merged greedy inference demonstrates the importance of identifying crucial elements at the beginning of page loading. Thus, we combine the predictive element region forest mechanism and dependency-merged greedy inference together to test their impacts. As shown in Figure 16 (③), by taking the contribution to the visual completeness of each object into account, dependency-merged greedy inference generates the (nearly) SI-optimal page load scheduling, which guides object loading and significantly improves SI by 25.93% on average. Note that neither the single use of predictive element region forest nor the single use of dependency-merged greedy inference can have such a high improvement. Therefore, the combination of these two mechanisms is critical to the performance of SipLoader.

Efficiency and optimality. We next compare the efficiency and the optimality of dependency-merged greedy inference with the topological sort-based approach. Figure 17 depicts the average computation time of them on web pages with a small number of objects. When the object number reaches 12, the topological sort-based approach bears an average computation time of 1,261 ms, significantly delaying the page loading process. Moreover, when the number of objects exceeds 14, the computation time of the topological sort-based approach even makes further comparisons hard to be conducted. In contrast, the polynomial complexity of dependency-merged greedy inference allows us to quickly obtain the nearly SI-optimal scheduling. For web pages with tens of objects, it only consumes no more than 1-ms computation. Also, as shown in Figure 18, for all of the 300 corpus web pages, we can obtain the nearly SI-optimal scheduling in 12.08 ms on average.

To evaluate the optimality of the baseline scheduling created by dependency-merged greedy inference, we first choose landing pages from the Alexa top 2,000 sites whose SI-optimal page load scheduling can be calculated by the topological sort-based approach in 20 minutes. Finally, we have 134 pages' SI-optimal schedulings as the ground truth. We then compare them with the corresponding baseline schedulings generated by dependency-merged greedy inference, but observe no substantial difference in the optimality of these two types of scheduling. The NP-completeness of the SI-optimal scheduling problem prohibits us from further evaluating our greedy inference algorithm. Nevertheless, considering the uncertain environments in most page loading processes, we feel that the baseline generated by our greedy inference algorithm is sufficient since the widely-existed uncertainties make a theoretically optimal baseline scheduling unnecessary, and it will be more important to actively react to these uncertainties during the real scheduling process (which is also the key idea of the original predictive-reactive scheduling framework [37]).

Event-driven reactive co-scheduling. We further evaluate the impacts of the event-driven reactive co-scheduling mechanism. As shown in Figure 16 (④), when it is enabled along with the other two mechanisms, the SI improvement increases from 25.93% to 32.24%. This illustrates that page loading processes are oftentimes subject to considerable uncertainties, causing the "anticipated" SI-optimal scheduling to easily become suboptimal. Thus, reactively repairing the baseline scheduling during page loading can provide additional benefits.

6.4 Additional Results

Benefits on other metrics. Although SipLoader mainly focuses on improving SI, it actually benefits other page load metrics as well. For instance, SipLoader benefits Above-the-Fold Time since it gives higher priorities to loading crucial elements and thus web pages' above-the-fold sections can be rendered more quickly. Figure 19 shows the cumulative distribution of the Above-the-Fold Time of the 300 corpus pages with/without SipLoader. As shown, the average Above-the-Fold Time improvement is 26.08%. Moreover, SipLoader improves Page Load Time (shown in Figure 20). This is because SipLoader receives the dependency graph at the beginning of page loading, which allows it to proactively fetch web page objects as long as the network pipe is not busy, rather than issues network requests until the objects are really requested by the page.

Additional traffic overhead. As SipLoader requires apriori knowledge such as the dependency graph and the predictive element region forest to perform page load scheduling, we also measure the extra traffic overhead incurred by the transmission of these data. Among tested pages, SipLoader only consumes 12.32 KB additional data usage on average, which is negligible compared with usually MB-level traffic overhead of common web pages' loading processes.

Server-side overhead. SipLoader incurs negligible overhead to the server in the phase of web page preloading and rewriting owing to the design that takes advantage of efficient computational geometry algorithms. For an average web page, it only needs ~30 seconds to finish all the jobs, indicating that page changes on a time granularity of >30 seconds can all be handled well. The average memory usage during the preloading and rewriting of a page is 386 MB, most (87%) of which comes from the headless puppet browser [16]. Also, SipLoader does not incur extra CPU/memory overhead on web servers during the real page loading, as the client requests the rewritten version of the page as normal pages, and the page load scheduling is performed entirely at the client browser.

Case study. Finally, we take the landing page of MSN as an example to reveal the real benefits of SipLoader on user experience. Figure 21 shows the visual progress of page loads with the page's default loading scheme and the (nearly) SI-optimal scheduling generated by SipLoader. At the beginning of page loading, there is no significant difference between the two loading schemes. This is because both of them obey the same dependency policy of the page, within which some objects must be loaded strictly ahead of the others, such as scripts that initialize global variables.

However, at around 1,800 ms, the default page loading scheme starts to evaluate scripts to load several small and non-static icon images (*e.g.,* icon A and icon B in the figure) that contribute little to the visual completeness but occupy the network connection slots for a long time (①). As a result, requests for subsequent static and large images have to wait until the browser receives the preceding small icons (②). In contrast, SipLoader determines the nearly SI-optimal loading orders for web page objects, and therefore loads the large, static images (that have no precedent dependencies) first (③) and then evaluates the scripts for loading small icons. Also, when uncertainties cause objects to be received later than expected, SipLoader reactively repairs the scheduling by selecting objects that have no precedent dependencies to be fetched or evaluated first (④). In this way, uncertainties hardly block the loading process, and user-perceivable page elements are oftentimes rendered with higher priorities, which provides a better experience for users during page loading.

7 RELATED WORK

Page load performance. The performance of web page loading has attracted increasing attention from both industry and academia. For example, Lighthouse [17] is an automated tool integrated in Chrome DevTools [13] for web page developers to measure web page performance, debug page load processes, and pinpoint the performance bottleneck. Also, a recent study in [46] builds a testbed to measure mobile web page load performance. They characterize the unique factors that can cause poor performance to mobile page loading, including the computational power of mobile devices, the different critical path from the desktop-version pages, *etc.*

In this paper, we conduct a measurement study to uncover the uncertainties during page loading, and thereby provide useful insights on the design of web page performance optimization methods.

Web page accelerators. In the past ten years or so, there have been plenty of techniques for optimizing web page loading, which can be mainly classified into three categories: 1) dependency-based page load scheduling, 2) snapshot-based optimization, and 3) user-oriented QoE enhancement.

WProf [61] resolves the dependencies of object loading during the real page load process. Scout [48] uncovers hidden dependencies among objects by tracking fine-grained data flows. Based on such detailed dependency information together with observations about real-time network conditions, Polaris [48] schedules the fetching of a page's objects by dynamically prioritizing those in the critical loading path. Moreover, Vroom [57] adopts a server-aided dependency resolution scheme to make the dependency-based page load scheduling both efficient and secure. On the other side, strategies on placing high-latency objects in CDN cache hierarchies are also studied in [55].

For snapshot-based optimization techniques, the most notable one among them is Prophecy [49]. The Prophecy server precomputes the final states of JavaScript and DOM tree for a page, which are transformed into write logs for the client to quickly initialize the page. Another snapshot-based accelerator, Shandian [62], preloads the web page on the server side and migrates the page state to the client with inefficient loading processes removed. In addition, the study in [63] provides a tool that allows developers to identify nondeterminisms [63] of JavaScript executions, so that the correctness of snapshot-based page loading can be effectively guaranteed.

Recent studies also take the first step to optimizing page loading from the aspect of users' subjective feelings. For instance, Klotski [29] generates page load scheduling that prioritizes high-utility contents for arbitrary users. WebGaze [40] prioritizes objects that are more visually interesting to users. Besides, adPerf [54] identifies pages' third-party ads that can cause user experience issues.

However, all the above efforts use page load metrics to retrospectively evaluate the benefits of page load schemes, rather than use them as active guidance to direct page loading. In contrast, we proactively take an advanced page load metric, Speed Index, into the design and implementation processes of page load schemes to enable metric-guided page load acceleration.

8 CONCLUSION

We present SipLoader, an SI-oriented page load scheduler that proactively takes SI as guidance to page loading. Based on our novel cumulative reactive scheduling framework, SipLoader "repairs" the anticipated (nearly) SI-optimal scheduling when scheduling uncertainties deriving from network, browser execution, and user-side viewport size actually occur during page loading. Evaluations show that SipLoader improves the median SI by 41%, outperforms the state-of-the-art solutions by 1.43×–1.99×, and incurs limited computation and traffic overhead.

Despite the above merits, SipLoader still bears two major limitations at the moment. First, the mechanism of predictive element region forest does not well accommodate personalized or highly dynamic pages whose layouts can vary greatly across different page loads. Second, SipLoader may cause performance degradation in a small part of simple pages because scheduling page loading incurs extra overheads, which cannot be compensated by the benefits gained on these simple pages. We are exploring practical ways to overcome these limitations.

ACKNOWLEDGMENTS

We thank our shepherd, Gareth Tyson, and the anonymous reviewers for their valuable comments and suggestions. Also, we thank Jian Yang for his careful and comprehensive measurement study in the early stage of this research. This work is supported in part by the National Natural Science Foundation of China (NSFC) under grants 61822205, 61902211, 61632020, and 61632013.

REFERENCES

[1] 2007. Liquid Layout. https://www.w3.org/WAI/GL/WCAG20/WD-WCAG20-TECHS-20071102/G146.html.

[2] 2016. Speed Index: A Key Metric for the User Experience. https://www.fasterize.com/en/blog/speed-index-a-key-metric-for-the-user-experience/.

[3] 2016. Speedline. https://github.com/paulirish/speedline.

[4] 2017. Understanding Speed Index. https://www.catchpoint.com/blog/speed-index.

[5] 2019. How to Use Google's Speed Index to Improve Wordpress Performance. https://wpmudev.com/blog/speed-index-wordpress/.

[6] 2019. Reduce the Scope and Complexity of Style Calculations. https://developers.google.com/web/fundamentals/performance/rendering/reduce-the-scope-and-complexity-of-style-calculations.

[7] 2020. Responsive Web Design Basics. https://web.dev/responsive-web-design-basics/.

[8] 2020. Time to Largest Paint. https://web.dev/lcp/.

[9] 2021. Alexa Top Sites. https://www.alexa.com/topsites.

[10] 2021. Amazon. https://www.amazon.com/.

[11] 2021. BBC. https://www.bbc.com/.

[12] 2021. Beautiful Soup Documentation. https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[13] 2021. Chrome DevTools. https://developer.chrome.com/docs/devtools/.

[14] 2021. Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol/.

[15] 2021. Elements' Bounding Rectangle. https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect.

[16] 2021. Getting Started with Headless Chrome. https://developers.google.com/web/updates/2017/04/headless-chrome.

[17] 2021. Lighthouse. https://developers.google.com/web/tools/lighthouse.

[18] 2021. MSN. https://www.msn.com.

[19] 2021. MutationObserver API. https://developer.mozilla.org/en-US/docs/Web/API/MutationObserver.

[20] 2021. Page Load Time. https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API.

[21] 2021. Pinterest. https://www.pinterest.com/.

[22] 2021. Puppeteer: Headless Chrome Node.js API. https://github.com/puppeteer/puppeteer.

[23] 2021. Speed Index. https://web.dev/speed-index/.

[24] 2021. Time to First Byte. https://developer.mozilla.org/en-US/docs/Glossary/time_to_first_byte.

[25] 2021. Time to First Paint. https://developer.mozilla.org/en-US/docs/Web/API/PerformancePaintTiming.

[26] 2021. Using Liquid Layout. https://www.w3.org/WAI/WCAG21/Techniques/general/G146.html.

[27] 2021. WebAssembly. https://webassembly.org/.

[28] 2021. WebPageTest Documentation. https://docs.webpagetest.org/metrics/speedindex/.

[29] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V Madhyastha, and Vyas Sekar. 2015. Klotski: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *Proc. of NSDI*. USENIX, 439–453.

[30] Chandra Chekuri and Rajeev Motwani. 1999. Precedence Constrained Scheduling to Minimize Sum of Weighted Completion Times on a Single Machine. *Discrete Applied Mathematics* 98, 1 (1999), 29–38.

[31] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT.

[32] Diego da Hora, Dario Rossi, Vassilis Christophides, and Renata Teixeira. 2018. A Practical Method for Measuring Web Above-the-Fold Time. In *Proc. of SIGCOMM Conference on Posters and Demos*. ACM, 105–107.

[33] Diego Neves da Hora, Alemnew Sheferaw Asrese, Vassilis Christophides, Renata Teixeira, Dario Rossi, Vassilis Christophides, Renata Teixeira, and Dario Rossi. 2018. Narrowing the Gap Between QoS Metrics and Web QoE Using Above-the-Fold Metrics. In *Proc. of PAM*, Vol. 10771. Springer, 31–43.

[34] Sebastian Egger, Peter Reichl, Tobias Hosfeld, and Raimund Schatz. 2012. "Time Is Bandwidth"? Narrowing the Gap Between Subjective Time Perception and Quality of Experience. In *Proc. of ICC*. IEEE, 1325–1330.

[35] Qingzhu Gao, Prasenjit Dey, and Parvez Ahammad. 2017. Perceived Performance of Top Retail Webpages in the Wild. *ACM SIGCOMM Computer Communication Review* 47, 5 (2017), 42–47.

[36] Brian Groelinger. 2019. Blocking: The Web Performance Villain. https://blog.bluetriangle.com/blocking-web-performance-villain.

[37] Willy Herroelen and Roel Leus. 2005. Project Scheduling under Uncertainty: Survey and Research Potentials. *European Journal of Operational Research* 165, 2 (2005), 289–306.

[38] Byungjin Jun, Fabián E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. 2019. AMP up Your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *Proc. of MobiCom*. ACM, 1–14.

[39] Nikhil Kansal, Murali Ramanujam, and Ravi Netravali. 2021. Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly. In *Proc. of NSDI*. USENIX, 269–287.

[40] Conor Kelton, Jihoon Ryoo, Aruna Balasubramanian, and Samir R. Das. 2017. Improving User Perceived Page Load Time Using Gaze. In *Proc. of NSDI*. USENIX, 545–559.

[41] Ronny Ko, James Mickens, Blake Loring, and Ravi Netravali. 2021. Oblique: Accelerating Page Loads Using Symbolic Execution. In *Proc. of NSDI.* USENIX, 289–302.

[42] Eugene L Lawler. 1978. Sequencing Jobs to Minimize Total Weighted Completion Time Subject to Precedence Constraints. In *Algorithmic Aspects of Combinatorics.* Vol. 2. Elsevier, 75–90.

[43] Zhenhua Li, Weiwei Wang, Tianyin Xu, Xin Zhong, Xiang-Yang Li, Yunhao Liu, Christo Wilson, and Ben Y Zhao. 2016. Exploring Cross-Application Cellular Traffic Optimization with Baidu Trafficguard. In *Proc. of NSDI.* USENIX, 61–76.

[44] Shaghayegh Mardani, Ayush Goel, Ronny Ko, Harsha V Madhyastha, and Ravi Netravali. 2021. Horcrux: Automatic JavaScript Parallelism for Resource-Efficient Web Computation. In *Proc. of OSDI.* USENIX, 461–477.

[45] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proc. of NSDI.* USENIX, 879–894.

[46] Javad Nejati and Aruna Balasubramanian. 2016. An In-Depth Study of Mobile Browser Performance. In *Proc. of WWW.* ACM, 1305–1315.

[47] Javad Nejati and Aruna Balasubramanian. 2020. WProfX: A Fine-Grained Visualization Tool for Web Page Loads. *ACM Human-Computer Interaction* 4, EICS (2020), 1–22.

[48] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-Grained Dependency Tracking. In *Proc. of NSDI.* USENIX, 123–136.

[49] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proc. of NSDI.* USENIX, 249–266.

[50] Ravi Netravali, Vikram Nathan, James Mickens, and Hari Balakrishnan. 2018. Vesper: Measuring Time-to-Interactivity for Web Pages. In *Proc. of NSDI.* USENIX, 217–231.

[51] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proc. of ATC.* USENIX, 417–429.

[52] Ravi Netravali, Anirudh Sivaraman, James Mickens, and Hari Balakrishnan. 2019. WatchTower: Fast, Secure Mobile Page Loads Using Remote Dependency Resolution. In *Proc. of MobiSys.* ACM, 430–443.

[53] Joseph o'Rourke. 1998. *Computational Geometry in C.* Cambridge University.

[54] Behnam Pourghassemi, Jordan Bonecutter, Zhou Li, and Aparna Chandramowlishwaran. 2021. adPerf: Characterizing the Performance of Third-Party Ads. In *Proc. of SIGMETRICS.* ACM, 37–38.

[55] Shankaranarayanan Puzhavakath Narayanan, Yun Seong Nam, Ashiwan Sivakumar, Balakrishnan Chandrasekaran, Bruce Maggs, and Sanjay Rao. 2016. Reducing Latency Through Page-Aware Management of Web Objects by Content Delivery Networks. In *Proc. of SIGMETRICS.* ACM, 89–100.

[56] V Ramasubramanian and Kuldip K Paliwal. 1992. Fast K-Dimensional Tree Algorithms for Nearest Neighbor Search with Application to Vector Quantization Encoding. *IEEE Transactions on Signal Processing* 40, 3 (1992), 518–531.

[57] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proc. of SIGCOMM.* ACM, 390–403.

[58] Manuel Serrano. 2018. Javascript AOT Compilation. In *Proc. of DLS.* ACM, 50–63.

[59] Ashiwan Sivakumar, Chuan Jiang, Yun Seong Nam, Shankaranarayanan Puzhavakath Narayanan, Vijay Gopalakrishnan, Sanjay G. Rao, Subhabrata Sen, Mithuna Thottethodi, and T. N. Vijaykumar. 2017. NutShell: Scalable Whittled Proxy Execution for Low-Latency Web over Cellular Networks. In *Proc. of MobiCom.* ACM, 448–461.

[60] Matteo Varvello, Jeremy Blackburn, David Naylor, and Konstantina Papagiannaki. 2016. EYEORG: A Platform for Crowdsourcing Web Quality of Experience Measurements. In *Proc. of CoNEXT.* ACM, 399–412.

[61] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proc. of NSDI.* USENIX, 473–485.

[62] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *Proc. of NSDI.* USENIX, 109–122.

[63] Jihwan Yeo, Changhyun Shin, and Soo-Mook Moon. 2019. Snapshot-Based Loading Acceleration of Web Apps with Nondeterministic JavaScript Execution. In *Proc. of WWW.* ACM, 2215–2224.

[64] Pengxiong Zhu, Keyu Man, Zhongjie Wang, Zhiyun Qian, Roya Ensafi, J. Alex Halderman, and Haixin Duan. 2020. Characterizing Transnational Internet Performance and the Great Bottleneck of China. In *Proc. of SIGMETRICS.* ACM, 69–70.

Received October 2021; revised December 2021; accepted January 2022