WProfX: A Fine-grained Visualization Tool for Web Page Loads

JAVAD NEJATI, Stony Brook University, USA
ARUNA BALASUBRAMANIAN, Stony Brook University, USA

Web page performance is crucial in today's Internet ecosystem, and Web developers use various developer tools to analyze their page load performance. However, existing tools cannot be used to identify the critical bottlenecks during the page load process. In this work, we design an online tool called WProfX that allows Web developers to visually identify bottlenecks in their page structure. The key to WProfX is that unlike existing Web performance tools, WProfX not only visualizes the page load activity timings, but also extracts the dependencies between the activities. Using the dependency structure, WProfX identifies the critical bottleneck activities. This lets a developer quickly identify why their page is loading slow and conduct what-if analyses to study the effect of different optimizations.

WProfX uses low-level tracing information exposed by most major browsers to extract the relationship between page load activities. The result is that WProfX works with most major browsers and newer browser versions. WProfX visualizes the page load process as a dependency graph of semantically meaningful Web activities and identifies the critical bottlenecks. We evaluate WProfX with 14 Web developers who perform three what-if analysis tasks involving identifying the page load bottleneck and evaluating the effect of a page optimization. All the participants were able to complete the tasks with WProfX, compared to less than 60% when using the popular developer tools available today. WProfX is currently being used by Web developers in a large telecom and at a Silicon Valley startup.

ACM Reference Format:
Javad Nejati and Aruna Balasubramanian. 2020. WProfX: A Fine-grained Visualization Tool for Web Page Loads. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 73 (June 2020), 22 pages. https://doi.org/10.1145/3394975

1 INTRODUCTION

Webpage load performance is critical to everyone in the Web ecosystem, with over 2 billion Web page hits through Google Search alone [2]. For example, Mozilla reduced the page load time of their landing page by 2.2 seconds, which resulted in 60,000,000 more downloads [3]. Small businesses are even more affected by poor Web page performance [6].

In response, the industry has developed several tools to help developers analyze and improve their pages [1, 4, 5, 10, 24]. Most of the developer tools visualize the page load process as a waterfall graph, showing the start and end time of each page load activity.

However, existing developer tools miss one crucial information: page load bottlenecks [45]. Web page load is a complex process that involves several activities that are inter-dependent. These dependencies create bottlenecks during the page load process since one activity can begin only after the dependent activity is completed. If the page load process is represented as a dependency

Authors' addresses: Javad Nejati, jnejati@cs.stonybrook.edu, Stony Brook University, USA; Aruna Balasubramanian, arunab@cs.stonybrook.edu, Stony Brook University, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
2573-0142/2020/6-ART73 $15.00
https://doi.org/10.1145/3394975

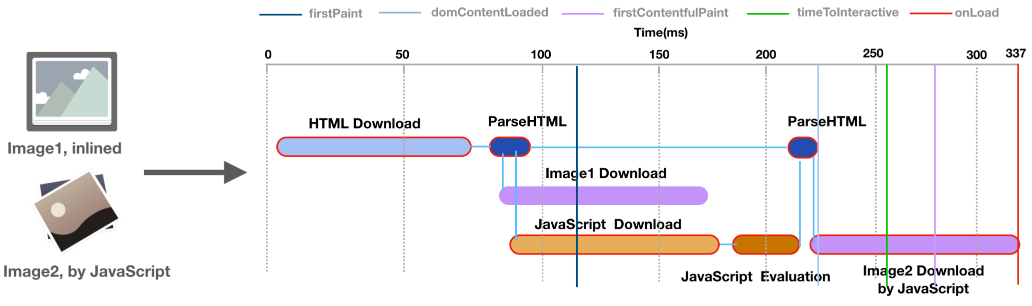

Fig. 1. An example WProfX visualization when loading a page consisting of an inlined image *(Image*1*)* and an image placed by JavaScript *(Image*2*)*. The visualization shows the Web activities during the page load process and critical activities. (rectangles with red borders)

graph of page activities, the critical bottleneck path is the set of activities that need to be optimized to improve page load time.

It is now well understood that identifying the critical bottleneck path is key to improving page performance [15, 37, 38, 44]. Existing works have used dependencies and critical path extensively to—reorder page activities to break dependencies and improve performance [37, 44], analyze the bottlenecks of mobile versus desktop pages [36], identify objects that should be cached [45], conduct what-if analysis [45], and many others.

The problem with existing works [36, 37, 44, 45] is they require significant changes to the browser source code to extract the dependency relationships and identify the critical path. This means that developers cannot use these tools for performance analysis, since such tools do not work beyond the one browser version on which the tool is implemented. On the other hand, industry developer tools work across browser versions, but they do not capture page dependencies or identify the critical path. For example, if a third-party image download on a page is on the critical path and affects page load performance, a developer cannot identify this bottleneck using existing developer tools.

In this paper, we present WProfX, an online performance analysis and visualization tool that scalably extracts dependency relationships and bottleneck path.

The key question in WProfX is: how can we extract the dependency relationships between page load activities without modifying the browser source code?. The core contribution of WProfX is to extract dependency relationships mentioned in [45] and [36] from low-level *trace* information that is exposed by all major browsers and can be obtained without browser changes. The result is that WProfX does not require changes to the browser source code and is portable across multiple browsers and browser versions.

The challenge is in mapping the low-level trace information into semantically meaningful page load activities. The low-level trace data is in the form of a sequence of discrete events dispatched at any given timestamp. For example, loading `youtube.com` generates over 80,000 trace events but corresponds to fewer than 100 page load activities. To this end, the WProfX rule engine filters unrelated trace events and aggregates relevant events into event clusters. For example the 80,000 `youtube.com`'s trace events are filtered in less than 1000 relevant trace events and then categorized in 75 clusters. WProfX then creates a template for each Web activity and fills the template using the information from the relevant traces. Finally, WProfX extracts the dependency between the Web activities using well known dependency rules [45].

We built an initial version of WProfX as stand-alone tool that visualize the page load activities and their relationships as a dependency graph, and identifies the critical bottleneck link. We distributed WProfX to 14 Web developers and researchers to get initial feedback. We received two key feedback. The first was that using a stand alone tool in an unfamiliar environment places burden on the developer. It is well known that learnability is one of the important aspect of usability [11, 19, 35, 40] and learning a new tool is burdensome [25, 30]. Instead, we re-implemented WProfX as an extension to a well-known developer tool. In our current prototype, we implement WProfX as a Chrome DevTool extension.

The second feedback was that WProfX provided fine-grained information that overwhelmed the users. We borrow ideas from previous research on coarse-grained visualization within a viewport [13, 21, 31, 33]. The result is a *compact mode* that allows users to visualize critical activities within a single viewport.

An example WProfX visualization is shown in Figure 1. The figure shows a simple page with two images, one directly embedded in the HTML page and the other fetched through a JavaScript [1]. WProfX shows the different Web activities during the page load process, extracts the dependencies (not shown here), and identifies the critical path shown in red. In this example, the second image load is on the bottleneck path, but the first image is not. This means, a developer can improve page performance by optimizing the second image load, for instance, by caching the image. But improving the load time of the first image will not improve the page load time.

To quantitatively and qualitatively evaluate WProfX , we asked 14 Web developers to complete three bottleneck-related performance tasks. We conducted a "within-subject" study where the developers finished each task with three Web performance tools: WProfX, Chrome DevTools, and PageSpeedInsights. We chose Chrome DevTools and PageSpeedInsights for comparison because they were the most popularly used Web performance tool in our survey.

The three tasks involved (i) identifying the bottleneck on a page, (ii) identifying a different bottleneck on a complex page with over 270 activities, and (iii) conducting a what-if analysis to predict the effect of the caching optimization on a page. All the participants were able to complete the task with WProfX for both Task #1 and Task #2. In contrast, only 20% of the participants were able to complete Task #1 using Chrome DevTools.

With Task #2, about 40% of the participants were able to complete the task with Chrome DevTools. But only 20% were able to complete the task with PageSpeedInsights.

The final task was performing a what-if analysis using WProfX. For this task, 91% of the participants were able to correctly identify the effect of the optimization using WProfX.

WProfX is currently being used at a large Telecom and at a Silicon Valley startup. WProfX is also in the process of being integrated into Google's LightHouse tool. We will release the tool and the WProfX code to the community.

2 VISUALIZATION TOOLS: RELATED WORKS

Our work builds on prior related works on user interfaces and visualization. We discuss these next. Tools specific to the Web are discussed in the next section.

Visualizing temporal data: There has been considerable work in the community on designing visualization tools for temporal data. WProfX is a browser based timeline visualizer similar to TimeLine Curator [20]. The goal of TimeLine Curator [20] is to use a natural language processing based system to replace manually extracting and formatting event data from source documents. This type of timeline-based visualization is commonly used to represent time-related events [16, 42]

[1] We use "JavaScript" to refer "Synchronous JavaScript" throughout this paper.

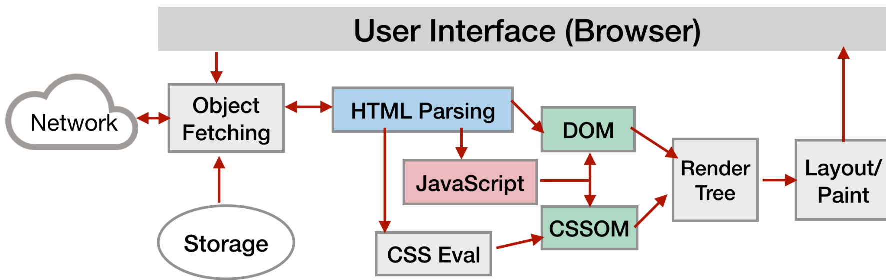

Fig. 2. Page load process inside a browser

There has also been several pioneering work on how temporal events should be visualized [47, 48]. WProfX borrows ideas from these existing tools, but in the context of Web performance.

Exploring Web page features: Unravel [26] and Telescope [27] show how Web pages can be reverse engineered so a developer can visualize and map the Web application code with a specific UI on the screen. The goal of both of these works is to help a developer learn new Web application features and explore the complex code base. In contrast, WProfX focusses on helping developers analyze Web performance and identify the critical bottlenecks during the page load process.

Relationship extraction: Representing the relationship between objects is key to visualizing complex operations. DA4Java [43] proposes a new way to simplify complex dependency graphs using simpler subgraphs in a complex programming environment. JaVis [34] is focused on a visualization and debugging environment for concurrent Java programs. In order to analyze user interactions on the web, Apaolaza et al. [12] define coarse-grained semantic events by mining low-level data.

Succinct representation: Succinct representation is key to the usability of a visualization tool. In three separate studies, LaToza [31] find that searching through code and traces for the purpose of debugging as well as exploring program control flow are common and often time consuming to answer. Bifröst et al. [33] also find that users can be overwhelmed with too much additional information when visualizing a complex process. We find the same to be true with Web performance tools (§8.3).

Geymayer et al. [21] and Baudisch et al. [13] introduce transparent windows to overcome hidden contents when windows are stacked or overlapped. Although they are addressing this issue in a window management context, WProfX uses a similar idea to represent large dependency graphs without overlap.

3 WEB PAGES AND CRITICAL PATH: BACKGROUND, EXISTING TOOLS, AND THEIR LIMITATIONS

3.1 Background: Web page loads and dependencies

We first describe how the page load process works on a browser as shown in Figure 2. The browser starts by fetching the main HTML page. After receiving the first chunk of HTML page, parsing starts. Every time the parser encounters a Web object to download such as a JavaScript, a Cascading Style Sheet (CSS), or an image, it fetches the object over the network. The objects are evaluated and added to the Document Object Model, or DOM [9]. DOM is an intermediate representation of the Web page. The renderer iteratively reads the DOM objects and renders the page.

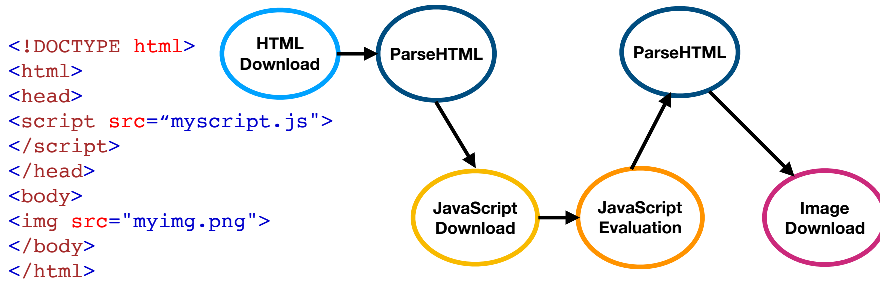

Fig. 3. A sample WProfX generated dependency graph.

The page load process can be broken down into five main activities: downloading which is a network activity, and parsing, evaluation, rendering and painting which are compute activities. Many of these activities can occur in parallel, but some activities can start only after a previous activity finishes. These dependencies are determined by the browser policies and browser implementation [45].

Based on the dependencies between the activities, one can draw a dependency graph, where a directed edge from one activity to another means that the second is dependent on the first. Figure 3 shows an example dependency structure for a simple Web page that has an embedded image and a JavaScript. Browser fetches the main HTML page and starts parsing the HTML. So the parsing is dependent on the HTML download. The parser encounters a *<script>* tag and downloads the JavaScript. Since both JavaScript and Parsing modify a common DOM data structure, the parsing stops until the JavaScript is downloaded and evaluated. When parsing resumes, the image tag is encountered. The browser now can download the image file. These are simple dependencies, but as we explain in §4.3, the dependencies can become complex.

3.2 Critical Bottleneck Path

The *critical path* is the longest path in the dependency graph such that reducing the latency of any activity not on the critical path cannot reduce the page load time.

Critical path analysis has been used to show which objects to cache; caching objects not on the critical path does not improve page performance [45]. By studying the critical path of a large number of Web page loads, WProfM [36] shows that compute activities are the bottleneck on mobile, but network activities are the bottleneck on desktops. This has resulted in follow up work on improve mobile Web performance by focussing on improving compute activities [17, 18].

Other works have used the critical path analysis and dependency graphs to perform what-if analysis and predict performance of Web services [32]. Recent works have shown how the order of the page load can be changed to break dependencies and improve performance [37] and user experience [15].

3.3 Limitation of existing tools

Unfortunately, Web developers cannot use the critical path analysis used by existing works [36, 37, 44, 45] to analyze their own Web pages. This is because, existing works require substantial changes to the browser source code to extract dependencies and identify the critical path. This means that to use these works to get the dependency graph, one needs to either port the browser changes to each version, which is tedious, or only stick to the browser version for which the system was originally implemented.

On the other hand, most major browsers provide commercial tools to analyze Web page performance that work for a large number of browser versions. These developer tools include—Chome DevTools [1], Google Lighthouse [24], Firefox Quantum [5], PageSpeedInsights [4], WebPageTest [10].

The problem is that existing DevTools do not capture the page load bottlenecks. Chrome DevTools, Quantum, and WebpageTest provide a waterfall graph showing the start and end time of each page load activities. However, they do not capture the relationship between the activities or provide a critical path. PageSpeedInsights and Lighthouse provide a score for the Web page based on how well they are designed. Both tools use only an *approximation* of the critical path to determine the scores.

In terms of granularity, PageSpeedInsights and WebPageTest only provide a high level score of the Web page and suggestions for optimizations. Chrome DevTools, Quantum, and WebpageTest are too fine-grained, providing details about all threads of the page loading process. In our survey of 14 Web developers (details in §8), 66% of the participants said that one of the limitation of the existing Web performance tools they use is the granularity. One of the participants said, "Tracking the page load process in Chrome DevTools Performance panel can get overwhelmingly time-consuming for larger Web sites".

4 WPROFX

The design goals of WProfX are:

- Visualize Web page load process at the granularity of *Web objects* that is semantically meaningful to a Web developer.
- Capture the relationship between the Web objects and visualize the page load bottlenecks
- Design WProfX without requiring browser instrumentation, only using information that browsers already provide.

WProfX is a Web performance tool built using information readily available from most browsers, rather than require extensive browser instrumentation. To this end, we use low-level tracing information that most modern browsers provide [8]. Listing 1 shows an example tracing tool exposed by Chrome. By leveraging existing tracing tools, porting WProfX to newer versions of the browser is straightforward. For example, WProfX works for Chrome versions from 45 and up (current Chrome version is 73), and can be easily ported to other browsers (§4.5).

4.1 WProfX overview

Figure 4 shows the building blocks of WProfX. WProfX gets the trace data from the browser programmatically. Then, the steps are: (i) Aggregation: This step involves mapping the fine-grained tracing information into semantically meaningful Web activities. WProfX does this by creating a template for each Web activity and getting information from the trace tools to fill out the template, (ii) Dependency extraction: This step identifies the dependencies/relationships between the identified Web activities, and (iii) Visualization: WProfX visualizes the page load process as a dependency graph and identifies the critical path.

We explain WProfX with respect to the Chrome browser for ease of exposition. We discuss how we generalize WProfX to other major browsers later in the paper.

4.2 Aggregating fine-grained traces into Web objects

To aggregate the low-level information available in the tracing tools, WProfX first defines the Web activities of interest. The Web activities are *Networking, Loading, Scripting, Rendering,* and *Painting*.

The challenge in WProfX is to aggregate information from the low-level tracing data to capture the Web page activities. Chrome's tracing tool [8] records *events* on a per-thread level. For example,

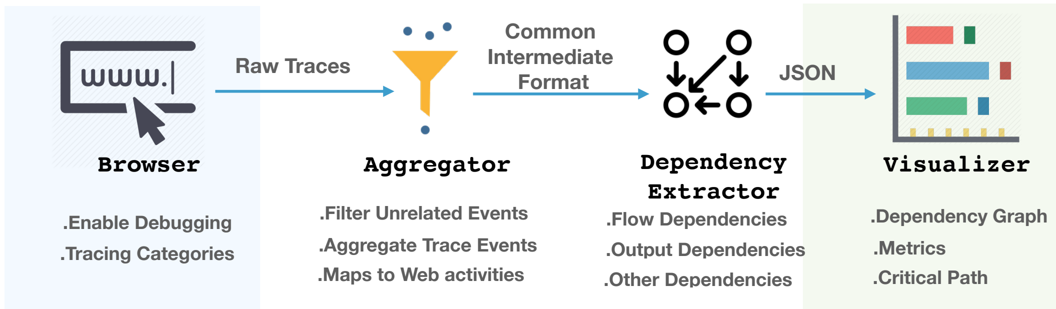

Fig. 4. WProfX extracts and visualize dependencies between activities based on browser traces.

Listing 1 shows the process ID (shown as "pid") followed by detailed information about the thread. For each event, the name, category, type, timestamp and ID of the process and the thread which dispatched that particular event recording are listed. The tracing tools of Firefox [5] is similar (§4.5).

```
{"pid":8030,"tid":1,"ts":405295098601,"ph":"I","cat":"disabled-by-default-devtools.timeline","
    name":"TracingStartedInPage","args":{"data":{"page":"0xeaf81d01e48","sessionId":"8030.1"}
    },"tts":81541,"s":"t"}
```

Listing 1. Sample Chrome *Trace event*

The first step is to filter the trace events and categorize them into different Web activities. However, information about each Web activity is spread across different trace events. One Web activity such as "Networking" corresponds to multiple trace events including sending the request, receiving the response, number of bytes sent and received, resolving the name to an IP address, among others.

Table 1 shows the template WProfX uses for each Web activity. The trace events are used to fill out the template for each activity and associates it with the corresponding Web object. The Web object is identified by the URL.

Figure 5 shows a simplified example of how a *Networking* activity can be aggregated from discrete trace *events*. The orange bar on the right side of the Figure 5 is a Web object being constructed from low-level tracing events which are shown on the left. WProfX collects start time, end time, mime type, and URL attributes from these trace data to build the Web object.

Table 1. Templates used to aggregate low-level trace events into their corresponding activities. One activity, such as "Networking" corresponds to multiple trace events, including DNS transaction, sending the request, receiving the request, etc. WProfX combines the trace information to find the start and end time and other attributes.

ResourceSendRequest, ResourceReceiveResponse, ResourceReceivedData, ResourceFinish, DNS_TRANSACTION, SOCKET_BYTES_RECEIVED, SOCKET_BYTES_SENT	Networking
ParseHTML, ParseAuthorStyleSheet	Loading
FunctionCall, EvaluateScript, v8.execute, v8.compile	Scripting
Layout, UpdateLayerTre, HitTest, RecalculateStyles	Rendering
CompositeLayers, Paint	Painting

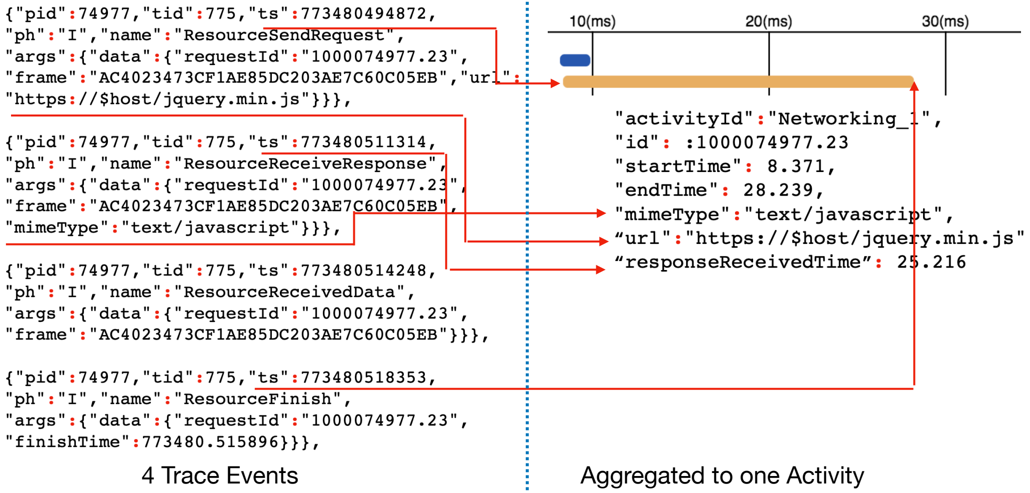

Fig. 5. An example of how WProfX aggregates trace *events* into a networking activity by filling the networking template. The resulting structure shows all the details for downloading the specific Web object.

Table 2. A subset of dependencies extracted by WProf [45], based on black-box testing techniques. The "–>" indicates the direction of dependency. For example, "a –> b" means, activity "a" is dependent on activity "b"

Dependency	Name	Definition
Flow	F1	Loading an object,→ Parsing the tag that references the object
	F2	Evaluating an object → Loading the object
	F3	Loading an object referenced by a JavaScript or CSS → Evaluating the JavaScript or CSS
Output	O1	Parsing the next tag → Completion of a previous JavaScript download and evaluation
	O2	JavaScript evaluation → Completion of a previous CSS evaluation
	O3	Parsing the next tag → Completion of a previous CSS download and evaluation

4.3 Relationship between Web objects

The next step is to extract the dependencies between the different activities. For this step, we rely on existing works [37, 45] that already extract the dependencies between different activities. We use these dependencies to encode relationship between objects and draw a dependency graph. Our goal then is to derive the relationships between the different Web objects by applying these dependency rules.

Table 2 shows a subset of the dependency policies extracted by WProf [45] for Chrome. The table categorizes the dependency policies into different categories, such as “Flow dependency" and “Output dependency". Dependency rules depend on the browser policies, but similar dependency rules are available for other browsers as well [45].

Flow dependencies Flow dependencies are self-evident; for example, loading an object depends on parsing the corresponding HTML tag. WProfX already computes starting and ending points of all

activities from trace events, and the URL of these activities are known. WProfX also extracts the specific activities when a tag is parsed. Using this, WProfX can create the dependency between parsing an object and downloading the object.

Dependencies also arise when an object is embedded within another object: for example, an image embedded within a JavaScript (F3 in Table 2). To extract this dependency, WProfX tracks the call chain across activities. The trace data provides enough information to create this call chain; for example, the trace data of a JavaScript event may encode that another object is embedded within that JavaScript. WProfX keeps track of this call and traces the embedded object. WProfX performs this call chain analysis recursively until all the embedded objects are accounted for.

Output dependencies:

Output dependencies are more complex and are a result of write conflicts. For example, JavaScript evaluation can modify the data structure called DOM while HTML parsing reads from DOM. The output dependency *O1* states that parsing is stopped until the JavaScript completes download and evaluation, to avoid HTML parsing and JavaScript evaluation run into a read-write conflict. WProfX orders all Web activities based on their timestamp. If the activity is a JavaScript download, all Web activities after that are blocked until JavaScript evaluation (another Web activity) is done.

Similarly, since CSS can modify styles of DOM nodes, execution of JavaScript is blocked until the style sheet is fetched (O2 in Table 2). To extract this dependency, WProfX detects all scripting download activities that their download time is before any CSS execution and updates the dependency graph to reflect that such JavaScript evaluations depend on previous CSS evaluation.

Additional dependencies can arise because of constraints in TCP/IP protocol stack and resource limitations. Browser scheduling policies and download priorities would also impact the order and timing of the fetched Web activities. In this section we only describe a subset of the dependencies, but WProfX extracts all dependencies uncovered by previous work [45].

4.4 Dependency graph, critical path, and visualization

Based on the extracted Web activities and their dependencies, WProfX creates a dependency graph. A dependency graph is a directed graph representing dependencies of different objects in respect to each other.

An example dependency structure is shown in Figure 6, captured in JSON format. In this example, two activities: a script activity and a loading activity are captured. The loading activity (ID: Loading_2) depends on the script activity (Scripting_0), as shown in the dependency block of the figure. To calculate the critical path, WProfX starts from the last Web activity and recursively finds the closest parent in the graph until it reaches to the the root (first activity).

Finally, the WProfX visualization tool is built on prior work on visualizing temporal events [20, 47, 48]. Past research has shown that representing temporal events as horizontal bars with related events placed together provides the best visual feedback to the users. Accordingly, the WProfX visualization tool draws the start and end time of each object horizontally using the D3.js [14] library. Related activities are placed in a horizontal sequence; for instance, loading and evaluating JavaScript are related activities. Similarly, the various HTML loading and parsing activities are related to each other and are placed in a single row. Activities that occur in parallel are vertically stacked.

4.5 Porting WProfX to major browsers

So far we described how WProfX works with Chrome, but WProfX generalizes to a large number of browsers. WProfX parses a trace file exposed by the browser and converts it to an intermediate

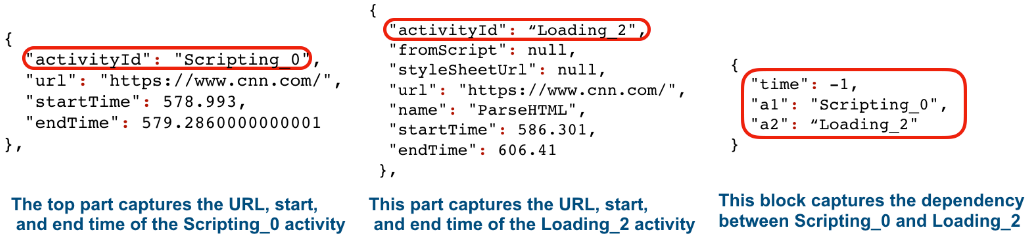

Fig. 6. The dependency relationship between different activities captured in a data structure.

representation of Web activities. The rest of the tool works on this intermediate representation and is independent of the browser. In other words, the last two blocks in Figure 4 work on the intermediate representation and are independent of the browser. Once the intermediate representation of the Web activities is extracted, porting WProfX to any browser is straightforward.

Major browsers fall under four browser engine categories: Blink (Chrome, Opera) , Gecko (Mozilla FireFox), Webkit (Safari) and the proprietary Microsoft EdgeHTML. WProfX works "as-is" with any Blink-based browser because the trace format is similar to that of Chrome. Figure 7 shows a snapshot of the WProfX visualization on Brave, Vivaldi, Opera, and Yandex browsers all of which are based on the Blink browser engine.

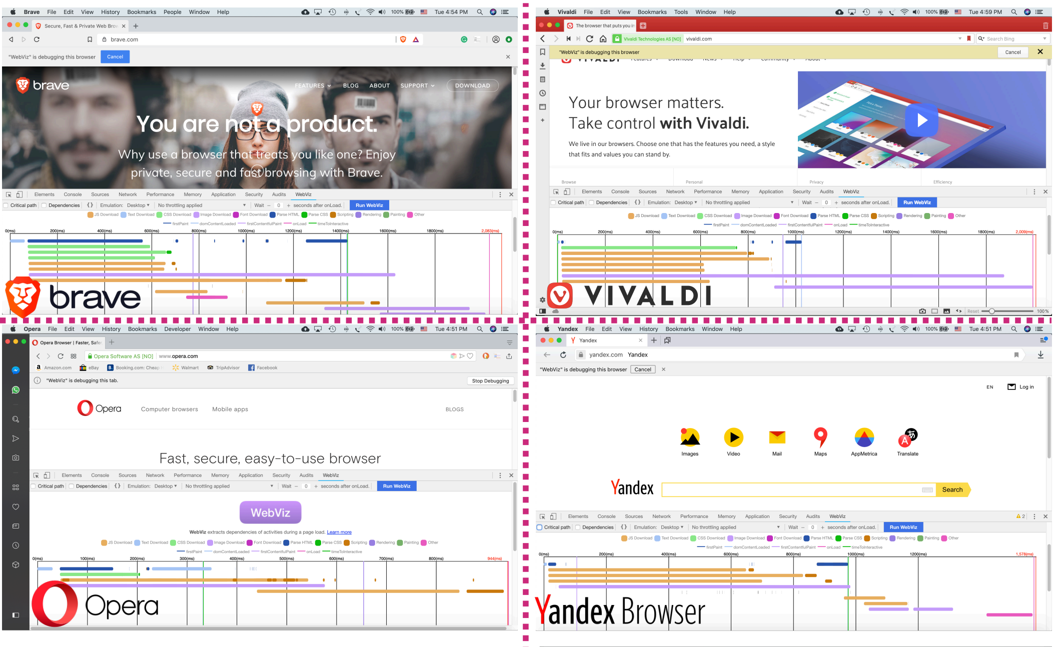

Fig. 7. A snapshot of WProfX running as-is on four browsers: Brave, Vivaldi, Opera, and Yandex.

WProfX has also been ported to Firefox and other Gecko-based browsers. Similar to Chrome, Firefox provides low-level tracing. Although the name of the variables and functions are specific to Firefox browser engine, we are able to extract the set of attributes necessary to construct a Web

Table 3. Demographics of participants in the survey.

Age		
<30 years	Between 30 and 35 years	>35 years
50%	28%	22%
Gender		
Female	Male	
14%	86%	
Location		
West Coast of USA	East Coast of USA	Outside of USA
15%	64%	21%

Table 4. Demographics of participants in the survey.

Web application experience		
>5 years	Between 2 and 5 years	<2 years
23%	42%	35%
Current Web performance tools used		
Chrome DevTools	PageSpeedInsights	WebPage Test
92%	42%	14%

activity and extract the dependency relations. It was recently announced that Microsoft browsers will be based on Chromium, which means WProfX can easily be ported to the Microsoft browsers as well. WProfX has not been ported to Webkit-based browsers because they do not yet provide sufficient low-level tracing information.

5 STUDY DESIGN

We administered a preliminary study and received feedback about the limitations. We then, designed the new tool based on the feedback (section 8). The second quantitative, task-based study was conducted on the improved version of WProfX.
For both studies, we designed Google forms to collect the feedback. We did not give any time limit but the average time reported is around 45 minutes.

Participants: Participants were selected by sending emails to Web researchers in academia and industry contacts. We contacted academic researchers who have written papers on Web performance in the past 5 years. We also emailed our industry contacts who are involved in Web technologies. Out of 26 emails sent, 14 people participated.

These 14 selected participants were across diverse demographics (Table 3) and with diverse Web experience (Table 4). 5 developers were from Silicon Valley, 4 developers working in the Telecom industry, and 5 were academic researchers.

Before we started the study, we asked the participants what tools they used for finding Web performance analysis. 93% of the participants used Chrome DevTools [1] for their Web performance analysis, 30% of then used PageSpeedInsights [7], and 14% used WebPageTest [10]. Participants did not mention any other Web performance tools.

For the preliminary study, we deployed the first version of WProfX and used it to build research artifacts. We also gave WProfX to 14 Web developers To these developers, we provided a video tutorial that was 4 minutes long describing the tool (similar to the video uploaded as supplementary material). We then asked them for their feedback.

For users not familiar with Chrome DevTools, we provided a tutorial similar to the WProfX tutorial. PageSpeed Insights is very simple to use, you enter a page's address and press analyze.

Then, it provides a performance score and a report based on a static checklist. We have not included WebPageTest since it is basically a trimmed down Web based version of Chrome DevTools.

6 PRELIMINARY STUDY

In the preliminary study, we provided the stand alone version of WProfX, We started by asking users about their familiarity with Web technologies and their experience with current tools. The participants pointed out 11 limitations for their current tools. Then, we introduced them to WProfX via a video tutorial and gathered their feedback in a 7-point likert scale for "ease of use", "ease of navigation", and "visual appeal" aspects of the tool.

Based on the feedback from the developers, we capture the lessons learned from the first version of WProfX.

Dependency relationship is useful: 13 of the 14 participants in our study found the bottleneck path and the dependency analysis to be extremely critical. The dependency analysis has also seen considerable use in research in the context of analyzing and improving Web page performance [15, 32, 37, 46]. We describe these in more detail in §2. Both of these indicate that dependency and critical path analysis is fundamental to performance analysis.

Tools need to be in a familiar environment: When asked, *'What characteristics are important for you in a Web performance tool?'*, 9 participants said that they would want the tool to be in a familiar environment. The first version of WProfX is a stand-alone tool that requires the Web developers to leave their development environment to analyze the Web page performance. The participants found this tedious.

Succinct representation is important: Even though WProfX aggregates trace information into a small number of Web activities, the visualization can still be overwhelming. Most modern Web pages have over 100 Web objects on an average, making the visualization span several pages. 5 of the 14 participants said they wanted a more succinct representation of the visualization, especially when analyzing large Web pages.

Performance metrics and programmability: Page load metrics have been a topic of increasing research [29, 39] and industry activity [23, 28, 41]. It is now well understood that the traditional Page Load Time metric is not sufficient to capture page load performance. Therefore, it is important for WProfX to capture these new performance metrics. In addition, 3 participants wanted an option to download the WProfX data, rather than having a visualization-only format. This would allow them to analyze the Web performance programmatically.

7 DESIGNING WPROFX BASED ON THE PRELIMINARY STUDY

Based on the lessons learned from the first study, we designed a new version of WProfX as described below.

7.1 Developer Tool Extension

Rather than a stand-alone tool, we implement WProfX as a DevTools extension. Developer tools are popularly used for Web performance analysis. In our own study, 92% of the Web developers used the Chrome developer tools for performance analysis (section 8). By adding WProfX as an extension, Web developers can use our visualization in a familiar environment.

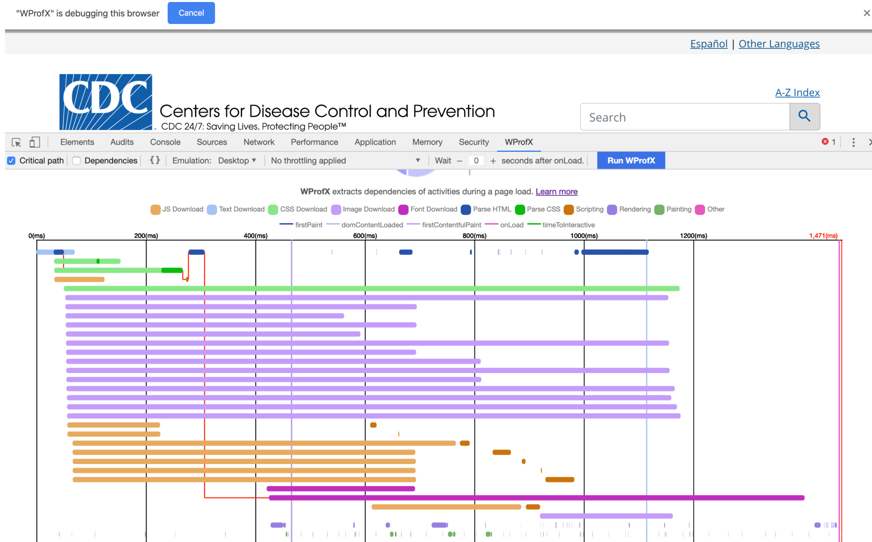

Fig. 8. WProfX visualization tool (original mode).

In our current prototype, WProfX is a Chrome extension. WProfX uses the browser's debugging protocol to capture the low-level trace programmatically, and then aggregates and visualizes the Web activities. An example WProfX visualization for `https://www.cdc.gov/` in the developer tools environment is shown in Figure 8.

7.2 Compact mode

Our earlier study showed that Web developers want a more succinct representation of the visualization. Related work in HCI and visualization show that compacting the visualization to fit the *viewport* of the user is critical to interacting with performance tools [13, 21]. To this end, we design a new compact mode of WProfX. Since the critical path is the key element of the Web performance, the compact model retains all the Web activities in the critical path and merges other activities of the same time together. A user can expand the merged activities if needed.

Figure 9 shows the compact mode of the visualization shown in Figure 8. The gaps in the critical path is because the browser switches to layout and painting tasks intermittently during the page load process (these tasks are also captured by the visualization). Typically the time taken for painting and layout is small enough to not be visible, but in this example the visualization is shown at the millisecond granularity.

7.3 Performance metrics

Apart from capturing the dependencies and the critical path, the tool also captures different page load metrics. The most commonly used page load time metric is *OnLoad*. *OnLoad* measures the time from when the browser starts evaluating the URL to when all Web page objects have been downloaded, parsed, and evaluated. Recently, some visual metrics have emerged, for example *SpeedIndex* [28]. SpeedIndex defines a Web page load in terms of when *most* of the content in the client's initial viewport is loaded. The newest set of metrics are the First Contentful Paint [41] and Time-to-interactivity [23] metric that define when the first paint event occurs and when a user can start interacting with the page, respectively.

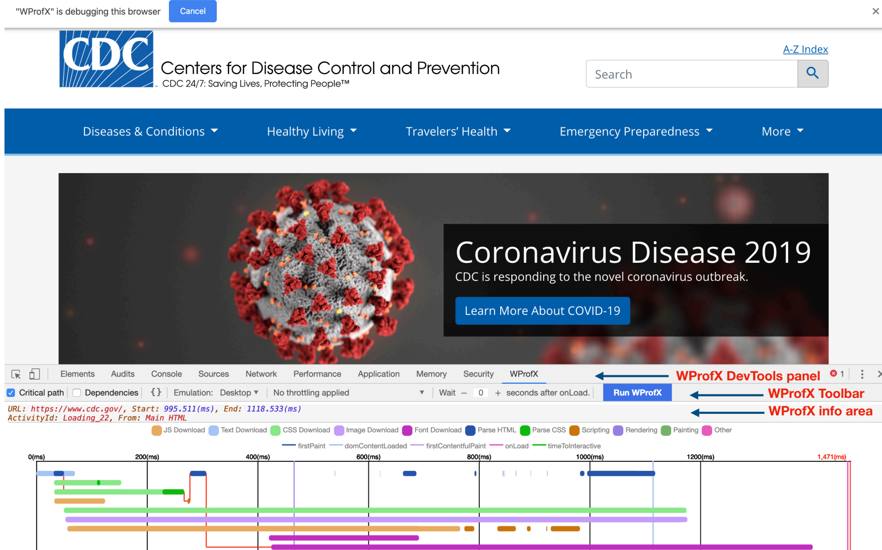

Fig. 9. WProfX visualization tool (compact mode). Shows only important activities compared to Figure 8 for the same Web site.

WProfX captures the First Contentful Paint and Time-to-interactivity in the visualization. Estimating the SpeedIndex metric requires capturing the video of the page load process because it is a visual metric. We are currently in the process of including the SpeedIndex metric as part of the tool.

In its default operation, WProfX loads Web pages in real time and analyses the page load process. In addition, the tool also allows emulating page loads on different devices (mobile and desktop) and under different network conditions.

8 WPROFX EVALUATION

The goal of our first user study was to get initial feedback on WProfX. After we receive the feedback and release the second version of WProfX, we conducted a second user study. The goal of this second user study is to compare the usefulness of WProfX compared to popular Web performance analysis tools. The study is approved by our Institutional Review Board.

Since Web pages are complex and diverse, the effect of applying an optimization on the Web page performance is unclear. To illustrate the effect of an optimization on different Web pages, we applied the *inlining* optimization to 4 websites. Inlining is a technique where the scripts, usually JavaScript and CSS, are included as part of the original HTML file, rather than as an external link. Figure 10 shows that out of 4 Websites, inlining helped only two of the Websites (lower is better). The same argument applies to other optimization techniques such as compression and caching. This mismatch is because the underlying bottleneck is different for different pages.

Since the key difference between WProfX and existing developer tools is with respect to page load bottlenecks, we design three tasks in terms of finding bottlenecks and conducting what-if analysis.

Third party contents are at the top of a page slow down list [22] (Task #1). Caching is the most widely used solution, usually applied via Content Delivery Networks, to improve the response time (Task #2) and as optimization can be page specific, Task #3 proposes a new way of evaluating an

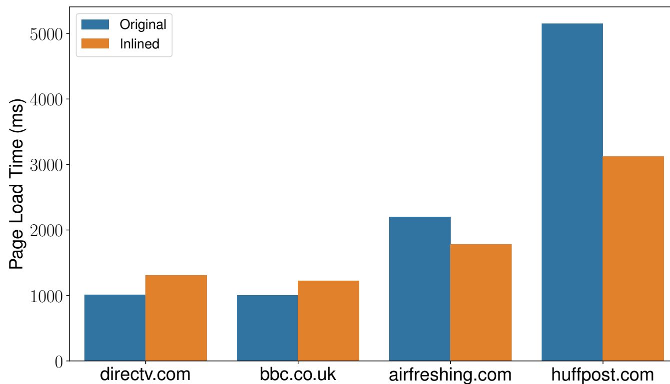

Fig. 10. Inlining, a popular optimization, does not uniformly improve the page load performance. PLT stands for Page Load Time, which is the time it takes to load the page. A given optimization helps certain pages and hurts others because the underlying bottleneck is different for different pages.

optimization using critical path. We use Yes/No questions when asking whether they could finish the tasks or not and a 5-point likert scale when asking questions such as "rate the usefulness of the tool". Finally we gathered the general comments from the subjects.

8.1 Evaluation Methodology

Task-Based study We conduct a "within-subject" study where each participant completes the three tasks using three tools: WProfX, Chrome DevTools, and PageSpeedInsights. We chose these two tools for comparisons because they were the most popular Web performance tools used by our participants. If the participants were unfamiliar with Chrome DevTools and PageSpeedInsights, we gave a tutorial on them.

Task #1 and Task#2 focus on identifying a bottleneck in a page load process in a relatively simple and complex page. Task#3 focuses on what-if analysis on how a Web page optimization will affect page performance.

To each participant, we describe the task to be done. We permute the order of the tools. The order is shown in Table 5 and Table 6 for the two tasks. The tasks revolve around finding the bottleneck. If the participant finds the bottleneck using the first tool, we still ask them to continue and find the bottleneck in the subsequent tools and rate the usefulness of all tools.

For the first two tasks:

- We measure the percentage of users that completed the task with each tool as a function of the order in which the tasks are presented.
- We ask the users "Rate the usefulness of the tool to complete the task, providing a rating of 1 to 5". 5 being most useful.

The third task only involves WProfX.

Free-form Case Study: After completing the tasks, we provide WProfX to the participants. Some of the participants used it for their own Web development. We present the different ways in which WProfX was used by the practitioners.

Table 5. Orders of tools being used and percentage of completion for Task#1

Number of Users	Tools order	Percentage
4	PageSpeedInsights, ChromeDevTools, WProfX	0%, 25%, 100%
5	ChromeDevTools, PageSpeedInsights, WProfX	20%, 0%, 100%
5	WProfX, ChromeDevTools, PageSpeedInsights	100%, 40%, 60%

Table 6. Orders of tools being used and percentage of completion for Task#2.

Number of Users	Tools order	Percentage
4	PageSpeedInsights, ChromeDevTools, WProfX	0%, 25%, 100%
5	ChromeDevTools, PageSpeedInsights, WProfX	100%, 40%, 100%
5	WProfX, ChromeDevTools, PageSpeedInsights	100%, 40%, 20%

8.2 Task #1: Finding the bottleneck

In the first task, the participants were asked to find information in the tool that showed why the Web page load was slow. We designed a sample Web site which embeds a Google analytics script which is widely used for data collection purpose. The use of third party services is growing at a rate of nearly 42% per year [22]. These resources can range from social sharing buttons (e.g. Facebook, Twitter), Video streaming embeds (e.g. YouTube), utility libraries to advertising and analytics scripts (e.g. Google ads and analytics).

The bottleneck in Task #1 is a large image download that is triggered by a third-party script. This additional image download is on the critical path because it is triggered from the script, although typically image downloads are not on the critical path.

Table 5 shows that, when WProfX is given as the first tool, 100% of participants identified the reason for the slow Web page. Interestingly, even after the participants identified the problem with WProfX, only 40% and 60% were able to find information for why the Web page load was slow when using ChromeDevTools and PageSpeedInsight respectively. This is because the bottleneck information cannot be easily found by users when using the other two tools. When ChromeDevTools and PageSpeedInsight were given as the first two tools, participants were able to find the reason for the slow Web page load a maximum of 25% of the time. With WProfX, participants identified the bottleneck 100% of the time, irrespective of order.

The participants could not easily find the bottleneck when using Chrome DevTools because the DevTools only provide the start and end time of different activities, but does not capture the relationship between the activities or the critical path. Figure 11 shows the visualization in Chrome DevTools and WProfX. WProfX adds the critical path that allows the participants to easily find that the image download is the bottleneck.

The problem with the PageSpeedInsights tool is that it provides high level information about how to improve the page load process, but the tool is static. Since the image download is embedded in a dynamic script, PageSpeedInsights is not able to identify this.

When we asked participants " Rate each tool's "Usefulness" for task#1" on a scale of 1 to 5, 83% rated WProfX 5, 8.3% rated 4 and 8.3% rated 3. The result for PageSpeedInsights are 25% 5, 25% 3, 33% 2, and 17% 1 and for Chrome DevTools 8.3% 5, 58.7% 4, 33% (Table 7)

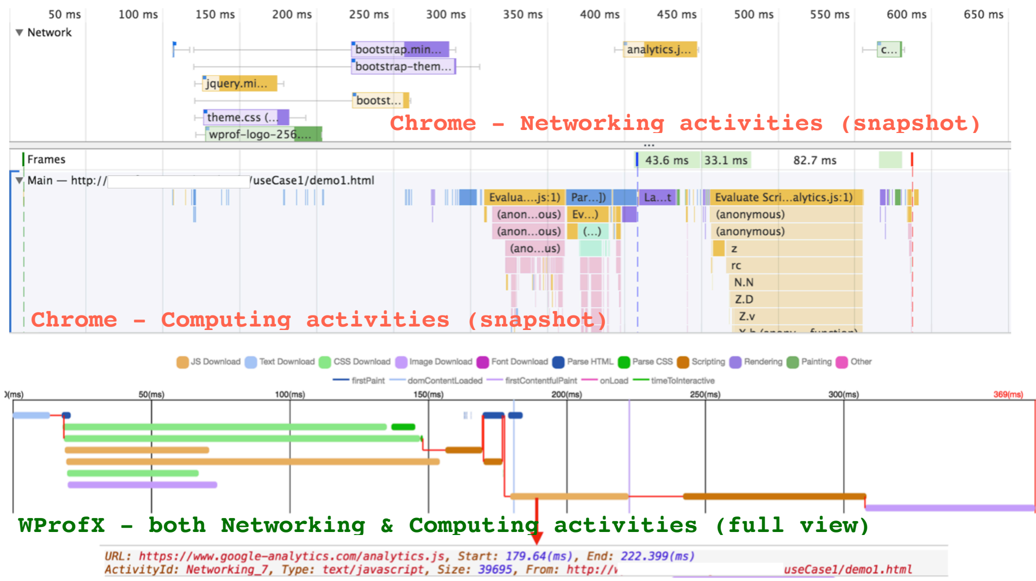

Fig. 11. Task #1: Chrome DevTools shows the Network and Computing activities in two different panels. Although the order and timing of each activity is depicted, no relationship between these activities are captured. WProfX, bottom of the figure, adds the critical path. From this it is evident that the image download is the bottleneck.

Table 7. Each tool's *Usefulness* rate for tasks 1 and 2.

Tool	WProfX					Mean	Variance	Chrome DevTools					Mean	Variance	PageSpeedInsights					Mean	Variance
Rate	5	4	3	2	1	-	-	5	4	3	2	1	-	-	5	4	3	2	1	-	-
Task#1	83%	8.3%	8.3%	-	-	4.6	0.8	8.3%	58.7%	33%	-	-	4	0.5	25%	-	25%	33%	17%	4.4	0.8
Task#2	75%	25%	-	-	-	4.8	0.2	8.3%	41.6%	33.3%	8.3%	8.3%	3	2	-	8.3%	33.3%	33.3%	25%	3	1.5

8.3 Task #2: Finding the bottleneck in a complex Webpage

In the second task, the participants were asked to analyze a large and complex Web page *(www.imdb.com)* and again find information in the tool that showed why the Web page load was slow. In this example, we injected a JavaScript to the original page. This JavaScript waits for one second (artificial delay) and returns. Since HTML parsing is being blocked by this JavaScript, the whole load process is affected. This artificial injection is a stand-in for complex scripts that may delay pages in real Websites.

Table 6 shows that if WProfX is presented as the first tool, 100% of the participants identified the bottleneck. As before, even though ChromeDevTools and PageSpeedInsights were presented after the user already found the bottleneck, only 40% and 20% of the users were able to find information in these tools about why the page load was slow. Interestingly, when ChromeDevTools was given as the first tool, all 5 participants were able to identify the bottleneck information in the tool. Overall though, ChromeDevTool was successful only 55% of the times.

One participants who did successfully identify bottleneck information in the tool using ChromeDevTools said: *"The output was cluttered and I had to spend half an hour to get useful information."*

Even with WProfX, it is difficult to analyze this page because of the large number of elements but the compact mode provides the information succinctly. Figure 12 shows the screenshot of WProfX

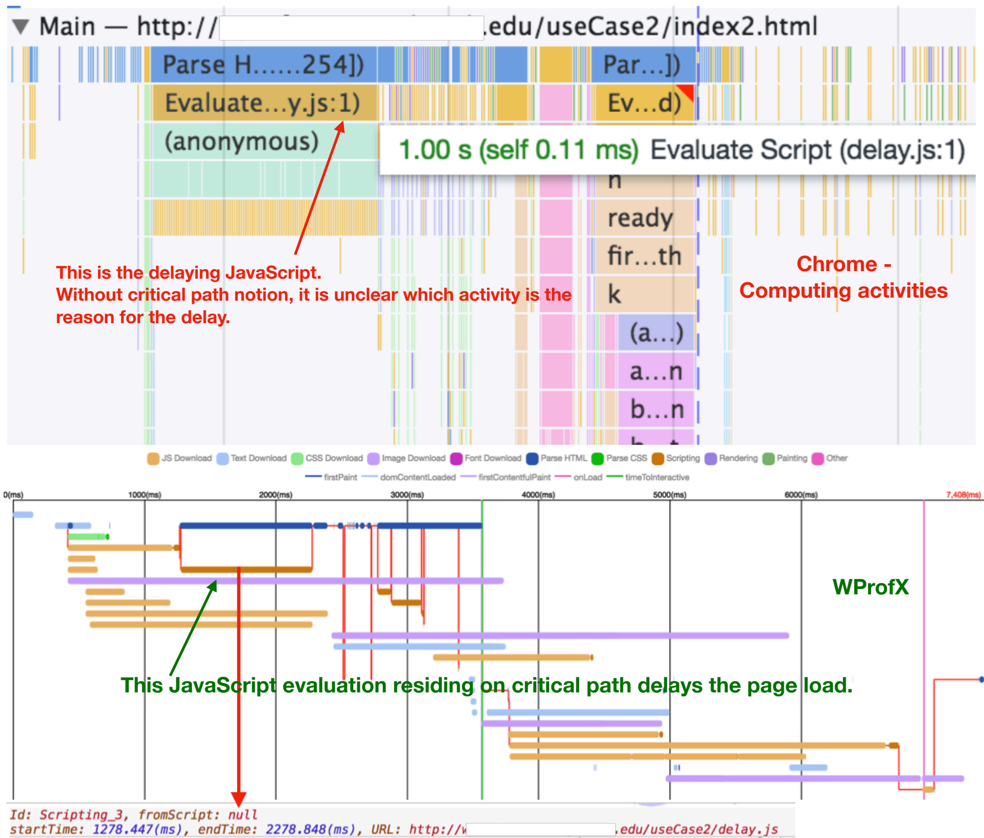

Fig. 12. Task #2: A large imdb page with an injected (blocking) JavaScript. The task was to determine what is causing the page to load slow. The bottleneck is the JavaScript.

and Chrome DevTools visualization for this page. PageSpeedInsights gives some useful guidelines about how to optimize this Web page. Specifically, a common optimization suggested by the tool is to cache images and remove blocking JavaScript. But it does not provide more insights. Table 7 shows that the participants found WProfX more useful in terms of a qualitative rating.

8.4 Task #3: What-if analysis

Finding the right set of optimizations for a given page is non-trivial because of the complexity and diversity of Web pages. An optimization that works well on one page may hurt performance on another page. When a Web developer applies an optimization to a Web page, they would like to see if that optimization helped to improve their page.

For this task, we do a what-if analysis where we cache 5 images on the same *imdb* website as the previous task. We then showed the page with and without the caching optimization and asked the participants to estimate the effect of the optimization. In Task#3, we analyze the usefulness of the

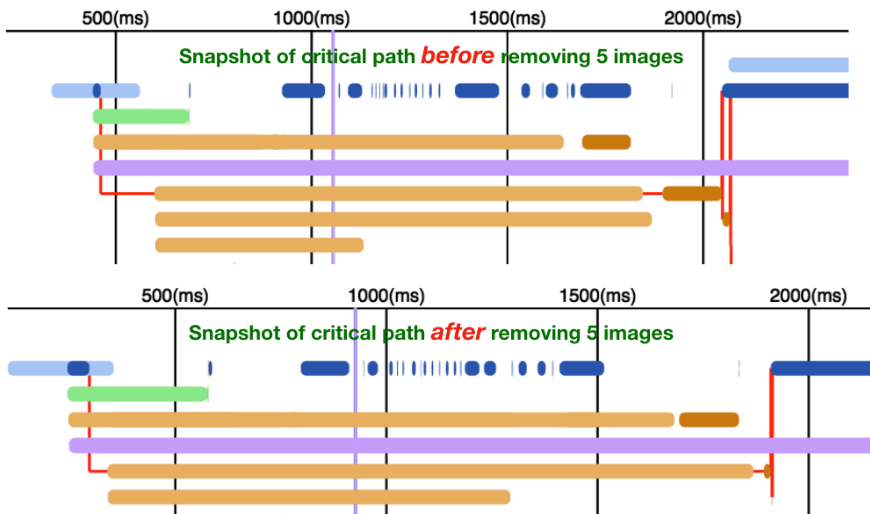

Fig. 13. Task #3: The participants were asked to study the effect of caching 5 images on a Web page. There was no improvement in the Web page load time due to caching because the images were not on the critical bottleneck path.

critical path feature of WProfX. Since this feature is only available in WProfX, we do not compare with alternate tools.

Figure 13 compares the activities on critical path during the image download phase before and after caching five images from the main Website. The dependency relationships of these two different runs are almost identical because the images that were cached are not on the critical path. 91% of the participants were able to identify why the critical path did not change and said that comparing the Web page loads before and after an optimization is an important feature to add to WProfX.

8.5 Free-form case study

Two participants, one from a Telecom industry and the other from a Silicon Valley startup, continued to use the tool for Web performance analysis. They provided feedback based on their experience with WProfX.

We have received the feedback two days (for case study #1) and 4 days (for case study #2) after we introduced the tool.

Case study #1 The Silicon Valley startup is using WProfX in their production environment. They are developing software products and an important part of every production workflow is the ability to continually test the new codes. They are integrating the WProfX output and metrics as part of their continuous integration process.

Case study #2 Another Web developer from a large telecom was able to find the bottleneck in their Web page. In this instance, the company was using a well-known Web framework that packs all the JavaScript in a large file. However, downloading this large JavaScript increased the page load time. They revised a strategy to separate the JavaScript files to download them on-demand to reduce the page load time.

Further, the Google LightHouse group is in the process of integrating WProfX into their tool.

9 LESSON LEARNED

Diminishing return of information overload : After developing the succinct representation (compact-mode) in WProfX, we observed that presenting only critical components that actually affect the page loading mechanics, significantly improves users ability to pinpoint the main culprit in a performance analysis task.

Fitting in the existing ecosystem: After our first preliminary study, we were asked to fit WProfX in a familiar environment. As a result, WProfX turned into a browser extension in the second study. After second study, we received feedback to make WProfX even more programmable to fit in automatic testing environments. These evolving feedback reflects the dynamic nature of today's continuous integration solutions and how it is dictating that any new software module should have a plan to fit in the current businesses' workflows. No software can survive by only relying on its standalone features anymore.

10 CONCLUSIONS

In this work, we present WProfX, a visualization tool that allows Web developers to perform Web performance analysis at a Web object level. Unlike existing Web page visualization tools, WProfX captures not only the page load activity timings but also the dependencies between the activities. Capturing the dependencies is crucial to finding bottlenecks, since dependencies play a critical role in determining the effect of an optimization on the page load process. WProfX does not require browser instrumentation, and can work with low-level tracing information provided by most modern browsers. We show how the WProfX tool is compared to other Web performance tools through a user study. The tool outperformed two existing popular tools both in terms of usability and performance in terms of finding bottlenecks.

However, one limitation in WProfX is that it uses the dependency policies extracted from browsers. If these policies change, then the WProfX dependency extraction engine will have to be modified accordingly.

11 ACKNOWLEDGEMENTS

This work was partially supported by the National Science Foundation, through the grant CNS-1566260. We gratefully acknowledge the anonymous reviewers who gave constructive feedback that greatly improved the presentation of the paper. We thank Jeannette Yu (University of Washington) who helped with the initial design of the WProfX visualization and Aditya Potdar (Stony Brook University) who improved the visualization code.

REFERENCES

[1] Chrome devtools. https://developers.google.com/web/tools/chrome-devtools/. (Accessed on 09/19/2018).
[2] Ericsson mobility report june 2015: http://www.ericsson.com/res/docs/2015/ericsson-mobility-report-june-2015.pdf.
[3] Firefox and Page Load Speed. https://blog.mozilla.org/metrics/2010/04/05/firefox-page-load-speed-%E2%80%93-part-ii/.
[4] Google Pagespeed Insights. https://developers.google.com/speed/pagespeed/insights.
[5] The new, fast browser for mac, pc and linux | firefox. https://www.mozilla.org/en-US/firefox/. (Accessed on 09/19/2018).
[6] Page Load Speed Impacts Buying Decisions. https://smallbiztrends.com/2019/02/page-load-speed-impacts-buying-decisions.html.
[7] PageSpeed Insights. https://developers.google.com/speed/pagespeed/insights/.
[8] The Trace Event Profiling Tool. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.
[9] W3C: Document Object Model. http://www.w3.org/DOM/.
[10] WebPagetest, Website Performance and Optimization Test. https://www.webpagetest.org.
[11] A. Abran, A. Khelifi, W. Suryn, and A. Seffah. Usability meanings and interpretations in iso standards. *Software quality journal*, 11(4):325–338, 2003.
[12] A. Apaolaza and M. Vigo. Assisted pattern mining for discovering interactive behaviours on the web. *International Journal of Human-Computer Studies*, 130:196–208, 2019.
[13] P. Baudisch and C. Gutwin. Multiblending: displaying overlapping windows simultaneously without the drawbacks of alpha blending. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 367–374. ACM, 2004.
[14] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.
[15] M. Butkiewicz, D. Wang, Z. Wu, H. V. Madhyastha, and V. Sekar. Klotski: Reprioritizing web content to improve user experience on mobile devices. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 439–453, Oakland, CA, 2015. USENIX Association.
[16] I. Cho, W. Dou, D. X. Wang, E. Sauda, and W. Ribarsky. Vairoma: A visual analytics system for making sense of places, times, and events in roman history. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):210–219, 2016.
[17] M. Dasari, C. Kelton, J. Nejati, A. Balasubramanian, and S. R. Das. Demystifying hardware bottlenecks in mobile web quality of experience. In *Proceedings of the SIGCOMM Posters and Demos*, SIGCOMM Posters and Demos âĂŹ17, page 43âĂŞ45, New York, NY, USA, 2017. Association for Computing Machinery.
[18] M. Dasari, S. Vargas, A. Bhattacharya, A. Balasubramanian, S. R. Das, and M. Ferdman. Impact of device performance on mobile internet qoe. In *Proceedings of the Internet Measurement Conference 2018*, pages 1–7. ACM, 2018.
[19] A. Dix. *Human-computer interaction*. Springer, 2009.
[20] J. Fulda, M. Brehmel, and T. Munzner. Timelinecurator: Interactive authoring of visual timelines from unstructured text. *IEEE transactions on visualization and computer graphics*, 22(1):300–309, 2016.
[21] T. Geymayer, M. Steinberger, A. Lex, M. Streit, and D. Schmalstieg. Show me the invisible: visualizing hidden content. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, pages 3705–3714. ACM, 2014.
[22] U. Goel, M. Steiner, W. Na, M. P. Wittie, M. Flack, and S. Ludin. Are 3 rd parties slowing down the mobile web? In *Proceedings of the Eighth Wireless of the Students, by the Students, and for the Students Workshop*, pages 18–20. ACM, 2016.
[23] Google. Time to interactive. https://github.com/WPO-Foundation/webpagetest/blob/master/docs/Metrics/TimeToInteractive.md. (Accessed on 11/31/2019).
[24] Google. LightHouse. https://developers.google.com/web/tools/lighthouse, 2019.
[25] T. Grossman, G. Fitzmaurice, and R. Attar. A survey of software learnability: metrics, methodologies and guidelines. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 649–658. ACM, 2009.
[26] J. Hibschman and H. Zhang. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pages 270–279. ACM, 2015.
[27] J. Hibschman and H. Zhang. Telescope: Fine-tuned discovery of interactive web ui feature implementation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 233–245. ACM, 2016.
[28] D. Imms. Speed index: Measuring page load time a different way. http://bit.ly/2dbuUpr, September 2014.
[29] C. Kelton, J. Ryoo, A. Balasubramanian, and S. R. Das. Improving user perceived page load times using gaze. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, 2017.

[30] B. Lafreniere and T. Grossman. Blocks-to-cad: A cross-application bridge from minecraft to 3d modeling. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, pages 637–648. ACM, 2018.
[31] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 185–194. ACM, 2010.
[32] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y.-M. Wang. WebProphet: automating performance prediction for web services. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI), 2010*.
[33] W. McGrath, D. Drew, J. Warner, M. Kazemitabaar, M. Karchemsky, D. Mellis, and B. Hartmann. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 299–310. ACM, 2017.
[34] K. Mehner. Javis: A uml-based visualization and debugging environment for concurrent java programs. In *Software Visualization*, pages 163–175. Springer, 2002.
[35] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.
[36] J. Nejati and A. Balasubramanian. An In-depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, pages 1305–1315, Montreal, Quebec, Canada, 2016.
[37] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association.
[38] R. Netravali and J. Mickens. Prophecy: Accelerating mobile page loads using final-state write logs. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 249–266, Renton, WA, 2018. USENIX Association.
[39] R. Netravali, V. Nathan, J. Mickens, and H. Balakrishnan. Vesper: Measuring time-to-interactivity for web pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 217–231, Renton, WA, 2018. USENIX Association.
[40] J. Nielsen. *Usability engineering*. Elsevier, 1994.
[41] S. Panicker. W3c paint timing working draft. http://bit.ly/2f2CGSk.
[42] F. Paternò, A. G. Schiavone, and P. Pitardi. Timelines for mobile web usability evaluation. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, pages 88–91, 2016.
[43] M. Pinzger, K. Graefenhain, P. Knab, and H. C. Gall. A tool for visual understanding of source code dependencies. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 254–259. IEEE, 2008.
[44] V. Ruamviboonsuk, R. Netravali, M. Uluyol, and H. V. Madhyastha. Vroom: Accelerating the mobile web with server-aided dependency resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 390–403. ACM, 2017.
[45] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystify page load performance with wprof. In *Proc. of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2013.
[46] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. How speedy is spdy? In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 387–399, Berkeley, CA, USA, 2014. USENIX Association.
[47] J. Zhao, S. M. Drucker, D. Fisher, and D. Brinkman. Timeslice: Interactive faceted browsing of timeline data. In *Proceedings of the International Working Conference on Advanced Visual Interfaces*, AVI '12, pages 433–436, New York, NY, USA, 2012. ACM.
[48] Y. Zhu, J. Yu, and J. Wu. Chro-ring: A time-oriented visual approach to represent writer's history. *Vis. Comput.*, 32(9):1133–1149, Sept. 2016.

Received July 2019; revised September 2019; accepted October 2019