What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling

BEHNAM POURGHASSEMI, University of California, Irvine, USA
ARDALAN AMIRI SANI, University of California, Irvine, USA
APARNA CHANDRAMOWLISHWARAN, University of California, Irvine, USA

Web browsers have become one of the most commonly used applications for desktop and mobile users. Despite recent advances in network speeds and several techniques to speed up web page loading such as speculative loading, smart caching, and multi-threading, browsers still suffer from relatively long page load time (PLT). As web applications are receiving widespread attention owing to their cross-platform support and comparatively straightforward development process, they need to have higher performance to compete with native applications. Recent studies have investigated the bottleneck of the modern web browser's performance and conclude that network connection is not the browser's bottleneck anymore. Even though there is still no consensus on this claim, no subsequent analysis has been conducted to inspect which parts of the browser's computation contribute to the performance overhead.

In this paper, we apply comprehensive and quantitative *what-if analysis* on the web browser's page loading process. Unlike conventional profiling methods, we apply *causal profiling* to precisely determine the impact of each computation stage such as HTML parsing and Layout on PLT. For this purpose, we develop COZ+, a high-performance causal profiler capable of analyzing large software systems such as the Chromium browser. COZ+ highlights the most influential spots for further optimization, which can be leveraged by browser developers and/or website designers. For instance, COZ+ shows that optimizing JavaScript by 40% is expected to improve the Chromium desktop browser's page loading performance by more than 8.5% under typical network conditions.

CCS Concepts: • **Information systems → Information systems applications**; • **Computing methodologies → Concurrent computing methodologies**; *Model development and analysis*;

Additional Key Words and Phrases: Web browsers, what-if analysis, causal profiling, page load time

ACM Reference Format:
Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran. 2019. What-If Analysis of Page Load Time in Web Browsers Using Causal Profiling. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 27 (June 2019), 23 pages. https://doi.org/10.1145/3326142

1 INTRODUCTION

Early web browsers only rendered static web pages with hyperlink documents but today's browsers are capable of loading web pages with animations, multimedia content, and JavaScript for user interactions. Moreover, trends in client-side web-applications since the introduction of *HTML5* and

Authors' addresses: Behnam Pourghassemi, University of California, Irvine, Irvine, CA, USA, bpourgha@uci.edu; Ardalan Amiri Sani, University of California, Irvine, Irvine, CA, USA, ardalan@uci.edu; Aparna Chandramowlishwaran, University of California, Irvine, Irvine, CA, USA, amowli@uci.edu.

2476-1249/2019/6-ART27 $15.00
https://doi.org/10.1145/3326142

Asynchronous JavaScript and XML (AJAX) have transformed web browsers into a critical platform for the end-user software stack.

Performance of the web browser is critical to its usability. An important metric for measuring performance is the *Page Load Time* (PLT). PLT is the time from the start of a user-initiated page request to the time the entire page content is loaded. PLT directly impacts user experience and even business revenue. Users may abandon a web page if it takes a long time to load or may even stop using a particular browser or website if it does not satisfy their desired performance. According to Google, 53% of mobile site visitors leave a page that takes longer than three seconds to load[1]. In 2016, AliExpress claimed that they reduced load time for their pages by 36% and recorded a 10.5% increase in orders[2].

There are two factors that contribute to PLT – (1) The time spent in *network activities* such as establishing a TCP connection or performing a DNS lookup. (2) The time spent in *computation activities* such as HTML parsing, applying CSS rules, etc.

Although there is a significant body of work on analyzing the source of performance bottlenecks in browsers, there is no consensus among them. On the one hand, researchers conclude that network activities are the primary source of performance overhead and several studies have investigated the effect of resource loading on the browser's PLT [3, 28, 42, 43]. Accordingly, various network infrastructure reconfiguration and client-side solutions have been proposed to diminish this source of overhead. Mitigating round-trip delay time, upgrading protocols along with the redesign of the browser's resource loading via prefetching, speculative loading, and smart caching are some of these techniques [13, 39, 43, 45].

On the other hand, more recent studies have implied that CPU-intensive phases such as HTML parsing and DOM manipulation have a more significant contribution to the PLT [32, 34, 37, 40, 45]. Correspondingly, researchers have attempted to improve the performance of different stages in the page rendering pipeline [36, 41, 47]. Browser developers also parallelize compute-intensive stages of the browser and fine-tune concurrency to mitigate page loading slow-down [27, 29, 32, 34, 35, 44].

In addition, browsers are getting more and more sophisticated in terms of both internal structure and code organization. Current browsers execute different computation and network activities on various threads and in some cases on multiple processes concurrently [12, 25, 31]. Inter-dependency between these activities establishes a critical path in the rendering process, which is highly complex to analyze [2, 26, 40, 46]. This raises two questions – (1) *What are the critical activities in the page loading process?* (2) *How much performance improvement would we realistically achieve by reducing these bottlenecks?*

In this paper, we employ *what-if analysis* on the page loading critical path to answer the above questions. Unfortunately, there is a paucity of literature on what-if analysis of computational activities on page loading process [37, 40, 42]. Furthermore, prior work is rooted in dependency extraction of the activities and static analysis of the dependency graph, which have restricted functionality since (1) these measurements are incapable of capturing all the existing inter-dependency between activities and (2) they do not take into account the dynamic behavior of the browser such as task scheduling and parser threading, and the dynamic behavior of content such as dynamically-generated object references in JavaScript [26, 38].

In order to analyze the browser performance, demystify the performance bottlenecks and evaluate their influence on PLT, we apply extensive and quantitative *what-if analysis* on the page loading process. Contrary to prior efforts, we use causal profiling [30], which indicates where

[1]https://www.thinkwithgoogle.com/marketing-resources/data-measurement/mobile-page-speed-new-industry-benchmarks

[2]https://edge.akamai.com/ec/us/highlights/keynote-speakers.jsp#edge2016futureofcommercemodal

the programmer should focus their optimization efforts and quantifies the potential impact of optimizations. The key idea behind causal profiling is to virtually speedup a selected line from the program at run-time and measure the impact of this acceleration on the total execution time. Causal profiling allows for the dynamic analysis of the critical path during application run-time. This method abstracts dependency extraction and subsequent dependency graph processing providing robust and adaptive what-if analysis of modern browsers. To apply causal profiling on web browsers, we build COZ+ on-top of the COZ profiler [24], the only implementation of the causal profiler (to the best of our knowledge). COZ+ virtually accelerates computation activities in the browser's page loading stages and computes the PLT improvement.

Contributions and findings. This paper makes the following contributions.

- We develop *COZ+*, an overhaul of the COZ profiler by adding multiple optimizations that target profiling overhead and redesign several modules to make causal profiling practically feasible and applicable to large applications. We further customize COZ+ for profiling the Chromium browser (an open-source version of Chrome) since it is currently the most popular browser for both desktop and mobile users [20][3].
- We perform comprehensive *what-if analysis* using *COZ+* on the major stages of the web browser's page loading process for the top 100 most popular web pages from Alexa Top 500 list [1]. Our analysis provides practical findings about browser performance (which in some cases contradicts prior work). For example, we observe that JavaScript contributes more to the page loading critical path than HTML parsing [37, 40] and by optimizing this stage by only 20%, the average PLT can improve by almost 5% on a desktop browser. This shows a considerable difference in comparison with the mobile browser (less than 0.5% [37, 42]).
- We examine the impact of different factors such as hardware, caching optimization, and network connection (varying network bandwidth and network delay) on the behavior of computation activities. Our results show that improving network connection and enabling caching have a small impact on PLT under typical network connections (i.e. bandwidth above 8 Mbps). From this, one can infer that computation is the bottleneck of current desktop browsers.

Our findings shed light on which stages the browser developers should focus their optimization efforts on to maximize overall performance. We observe that Scripting is the most influential stage (most "bang for the buck") followed by Styling and Layout irrespective of network bandwidth and delay. This enables focused optimization efforts to achieve performance goals. COZ+ can also benefit website designers. In addition to pinpointing the bottlenecks in their websites, COZ+ can assist in choosing the most suitable optimization from a list of available optimizations.

2 WEB BROWSERS

In this section, we first outline the browser's internal design and workflow. Next, we present the primary challenge in analyzing the performance of browsers due to the inter-dependency between the stages in the rendering pipeline. Finally, we provide an overview of the Chromium web browser, which we choose as the case study for the what-if analysis.

2.1 Browser architecture

Over the years, several browsers with different features, user interfaces, and security levels have come to the market. Regardless of their design and performance, they fundamentally share the same architecture and workflow for rendering web pages. The core software component of all major web browsers is a *rendering engine* (a.k.a. layout engine), which transforms the web page

[3]COZ+ is easily adaptable to other Webkit-based browsers.

plain content to the visual representation. Mozilla Firefox and Microsoft Internet Explorer (IE) have their own respective engines called *Gecko* [7] and *Trident* [17]. Microsoft's newer browser, Edge, uses *EdgeHTML* (a fork from Trident) [11]. The rest of the well-known browsers such as Google Chrome, Opera, and Safari are developed on top of the *Webkit* rendering engine [19].

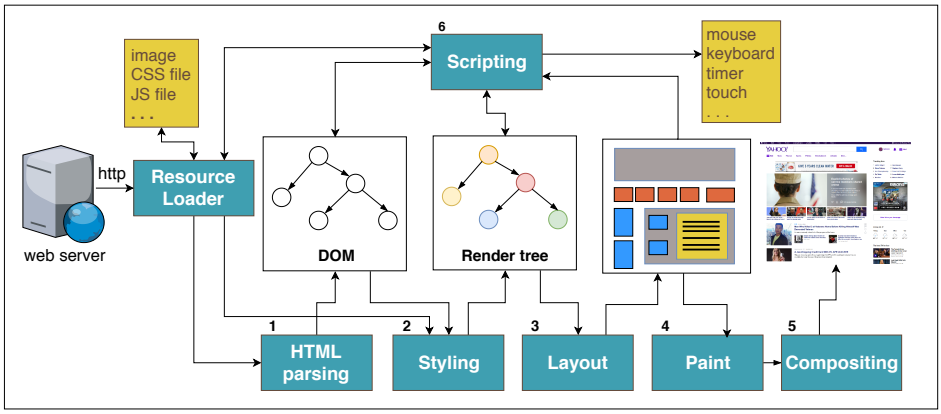


Fig. 1. The general workflow for loading web pages.

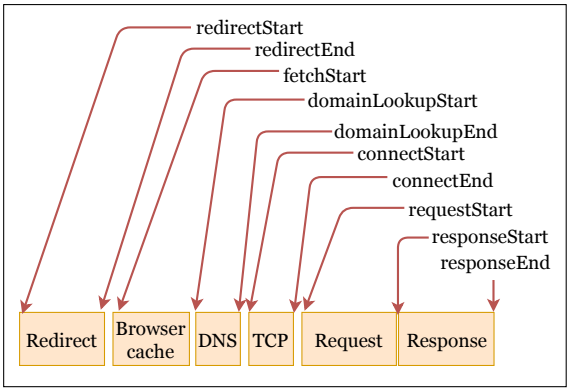


Fig. 2. Resource loading stack.

Figure 1 shows how browser engines load web pages. The process begins when the user submits a URL request to the browser interface. Immediately after that, the browser's *Resource Loader* initiates an HTTPS request to fetch the main HTML file from the web server. Typically, the Resource Loader downloads this file incrementally in order to hide part of the network delay with processing on received chunks. Figure 2 demonstrates the resource loader's internal workflow. When the first chunk of HTML is downloaded, the rendering engine starts parsing HTML tags and building the *Document Object Module* (DOM). DOM is an intermediate representation of the page content that is represented by a tree data structure. *HTML parsing* is the first computation stage in the rendering pipeline. During DOM construction, the HTML parser may request additional resources such as another HTML file, a CSS file, a JavaScript file, images, etc. For each request, the Resource Loader might apply DNS lookup and open a TCP port to download the object from the server or retrieve the object directly from the cache (Figure 2).

Among these resources, *Cascading Style Sheet* (CSS) files contain a set of rules that specify the format and attribute (e.g. font and color) of the page elements. The browser parses these rules and adds styling attributes to the DOM nodes. This stage referred to as *Styling* (stage 2) leads to the construction of another tree called the *render tree*. Nodes in the render tree are visual elements with the style characteristics that are ready to be displayed. In the third stage, *Layout*, the render tree is traversed to calculate the relative size and geometrical position of the elements on the screen. The fourth stage in the rendering pipeline is *Paint*, which is the process of mapping each visual part of the elements into pixels. Filling pixels is often done in multiple layers. At the end of the rendering pipeline, in *Compositing*, these layers are combined together to create a final view of the web page. *JavaScript* or generally *Scripting* (stage 6) is another computation stage in the browser that responds to the user interactions and handles the dynamic behavior of the web page. This stage consists of evaluating, compiling and executing the scripts and usually has a separate engine such as *V8* in Google Chrome [18] or *SpiderMonkey* in Mozilla Firefox [14]. JavaScript, like most of the other stages, has access to the DOM and can modify the DOM throughout the page loading process as seen from Figure 1.

2.2 Inter-dependency and critical path

There exist inter-dependencies between the above rendering stages during the page load process due to the fact that these stages constantly interact with the DOM. For example, JavaScript might use `document.write()` to insert/modify HTML content. As a result, Styling cannot proceed until DOM gets updated. To maintain coherency of DOM, access policies have been set by the browsers. For example, HTML parsing is blocked when it reaches the `<script>` tag. This tag (unlike `<async>` and `<defer>`) indicates that JavaScript might modify the DOM nodes. Therefore, the browser executes JavaScript code and then resumes HTML parsing. This ensures that the HTML parser accesses the updated DOM in the order that is declared in the context. In another scenario, JavaScript might change the styling format of some DOM nodes. This necessitates the browser to complete all ongoing CSS processes before servicing the JavaScript request. Wang et al. [40] analyze these dependency policies and categorize them into flow dependency, output dependency, lazy/eager binding, and resource constraints. All these dependencies restrict the browser's task scheduler to dynamically rearrange the order of stages, which in turn affects the PLT.

Figure 3 shows a concrete example of how these dependencies influence the PLT. In this example, the browser initially downloads the main HTML file, `a.html` and then starts parsing and constructing the DOM. The HTML parser encounters an external stylesheet, `b.css`, in line 4 (`<link>` tag) and starts loading it. Then, it parses a synchronize JavaScript tag, `<script>` in line 5, that references an external script, `c.js`. This tag blocks HTML parsing. Compiling and evaluating the external JavaScript resource, however, cannot proceed since the CSS file is still under evaluation. Due to this inter-dependency, JavaScript waits until `b.css` is loaded and evaluated. Once the CSS evaluation is done, the blocking script (`b.js`) is fully served and HTML parsing continues. Black arrows in the timeline in the bottom of Figure 3 represent dependencies between these activities. Ideally, if there were no dependencies, the three activities could be executed in parallel and the PLT would be determined by the slowest activity. However, in practice, the dependencies lead to a critical path as shown by the red dotted line. In this example, HTML parsing and parts of CSS and JavaScript are all on the critical path. It is easy to see that modifying this example (e.g. swapping line 4 and 5 in `a.html` to parse the script tag before the link tag or by manipulating the duration of the activities) will affect the critical path composition and consequently the page load time. These inter-dependencies between stages make characterizing PLT a daunting task.

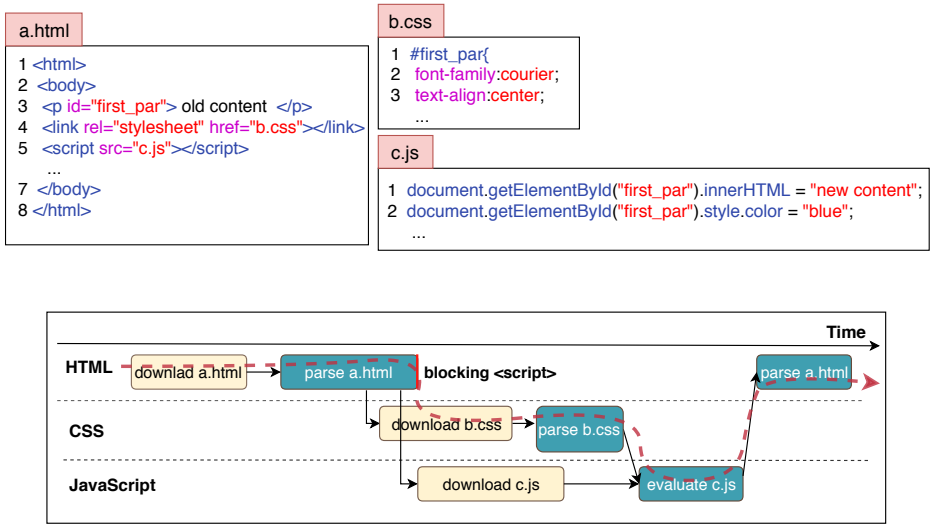


Fig. 3. (Top) An example to illustrate the dependencies between the different activities. (Bottom) Timeline showing page load activities. Black arrows represent the dependencies between activities and the red dotted line shows the page load critical path.

2.3 Chromium web browser

According to StatCounter [21], Chrome is the most popular web browser in use for both desktop and mobile devices. As of December 2018, it has 62.3% of the browser's market share and no other browser comes close. More specifically, Apple Safari has the second place with 14.7% of the market share and Firefox lags far behind with only 4.9% of the market share.

Architecture. The rendering engine of Chromium, Blink [23], is forked from the popular Webkit engine [19]. Chromium exploits process-per-site-instance architecture to protect the overall browser from crashes, glitches, or malware in web pages [6]. In this architecture, the main process, *browser process* runs the UI and manages tabs. One *renderer process* is created per web page instance. Chromium processes have multiple threads that handle page rendering, process communication, I/O operations, and so on, concurrently.

Most of the rendering stages like styling and layout run on the main renderer thread in the renderer process. However, parsing new HTML content gets its own thread similar to painting and compositing. JavaScript also runs on the main renderer thread, but with script streaming (new technique since Chrome version 41), JavaScript parses the scripts on a separate thread. JavaScript also interacts with the UI thread in the browser process to respond to user inputs. Moreover, it might spawn new threads called *web worker* threads to handle computationally intensive tasks in the background. In addition to these, resource loading and other network activities shown in Figure 2 are managed by I/O threads [12].

Challenge. Figure 4 shows a snapshot of the page loading timeline for `www.apple.com` obtained using the Chrome Trace Event Profiling Tool [16]. As we can see, multiple threads with different activities are involved in the page loading process. The web browser's complex architecture and inter-dependency between activities running on various threads make analyzing the critical path and page loading bottlenecks extremely challenging [2, 40, 46].

For *comprehensive what-if analysis* on modern web browsers with sophisticated code organization, web researchers and browser developers have to use a suitable tool. Conventional profilers for

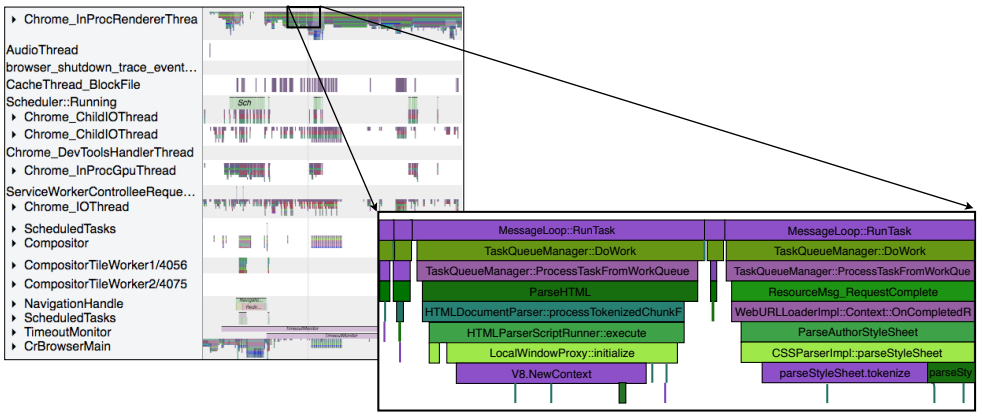


Fig. 4. Page load time for `apple.com` in April 2018. This timeline is obtained using the Chrome Trace Event Profiling Tool [16].

browsers like the *Chrome profiler* in *Chrome developer tools* use traces to record the duration of individual activity and do not quantify the effect of each activity on the PLT. Similarly, general-purpose profilers such as *gprof* [9] only rank the most influential functions based on how much time the program spends on them and do not report the potential impact of optimizing those functions. Although these profilers report an accurate timing of functions, relying exclusively on these statistics is not sufficient. For example, optimizing long JavaScript functions when the rendering process is waiting for a file to be downloaded will not improve the PLT. On top of this, the developer needs to have a deep understanding of the application source code to utilize these statistics for what-if analysis. To identify the bottlenecks and their potential impact on PLT, we need to consider the dependency between activities as well as the multi-threaded structure of the browser. In the next section, we discuss our methodology to address the above challenges.

3 COZ+: A HIGH-PERFORMANCE CAUSAL PROFILER

In this section, we first describe a novel profiling technique first introduced in [30] and how it can be leveraged to capture the dependencies in web browsers. Then, we present COZ+, a high-performance causal profiler capable of analyzing large complex software systems. We use Chromium (an open source version of the Chrome web browser) as our target case-study application.

3.1 Causal profiling

The key idea behind *causal profiling* [30] is to simulate the effect of speeding up a function by slowing down all other concurrent functions. Unlike traditional profiling methods, this technique *virtually speeds up* a selected line (e.g. a function call) at runtime and evaluates the subsequent variation in total execution time in a parallel application. At first glance, this might seem infeasible since the profiler needs to actively obtain information about the timing of all concurrent functions at runtime to decelerate them proportionally. However, the causal profiler uses a sampling method under the hood to frequently monitor threads and virtually speeds up the selected function by slowing down concurrent threads proportional to the sampling frequency. The proof of concept is presented in the original paper [30].

Figure 5 illustrates the concept of *virtual speedup* with a concrete example. The top timeline shows the execution of the original program with two threads running functions *A*, *B*, and *C*

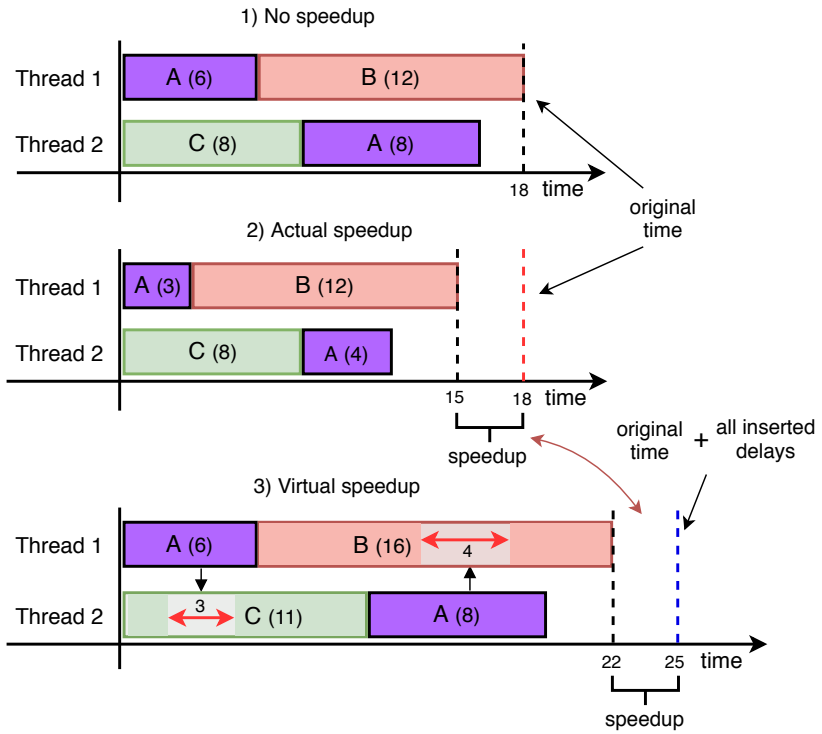


Fig. 5. Illustration of the concept of *virtual speedup* and *causal profiling*. The top timeline shows the execution of the original program with 2 threads running functions *A*, *B*, and *C* concurrently, the middle timeline corresponds to actually speeding up function *A* by 50%, and the bottom timeline shows the effect of virtually speeding up function *A* by 50%.

concurrently. The middle timeline demonstrates the effect of accelerating function *A* on the total execution time. The range indicated by *speedup* shows the *actual speedup* of the program after accelerating function *A* by 50%. The bottom timeline presents the effect of *virtually* speeding up A. Whenever A is executing, all other concurrent threads are paused for a certain amount of time depending on how much one intends to accelerate *A* (in this case, 50% of function *A*). The difference between the execution time of the program after virtual speedup and the original time of the program with all inserted delays (for this example, two slices of delays) gives the same speedup as actually optimizing *A* (middle timeline). In general, we can generate a what-if graph for function *A* by varying the amount of virtual speedup in *A* and plotting the corresponding program speedup.

A key insight of this paper is to *utilize causal profiling to apply what-if analysis to computation stages in a web browser*. There are multiple advantages in using a causal profiler over conventional profilers for web browsers. First is the support for multi-threaded applications with a complex dependency graph that have relatively short execution times. In this regard, it is a suitable candidate for page load time profiling since PLT takes a few seconds on average in modern web browsers. Second, we do not need to extract the dependency graph and apply graph processing to obtain the impact of components since all potential impacts of optimizations can be derived from multiple page loads for different speedups. Third, it captures the dynamic behavior of the application because it applies virtual speedups directly into the execution path at runtime.

3.2 COZ+

We introduce COZ+, a causal profiler capable of analyzing large and complex software systems such as the Chromium browser. COZ+ is built on top of COZ [24], an open source causal profiler (and the only such profiler available to the best of our knowledge). We originally intended to use COZ to profile the Chromium web browser. However, we soon learned that, while COZ works for simple benchmarks, it does not scale to large software systems due to several design and implementation issues. Therefore, we built COZ+, a comprehensive overhaul of COZ, which provides flexible profiling functionality for large applications as well as robust performance analysis capabilities for the Chromium browser. The COZ profiler has approximately 4k LOC and we modify/add around **1k lines** to build COZ+[4]. COZ+ is a standalone profiler and does not modify the browser source code. As a result, it can be applied to other web browsers as well and we plan to add support for other browsers in our future work.

Figure 6 shows the architecture of the COZ+ profiler broken down by the original design of COZ (shown in black) and our modifications (highlighted in red). The process begins from the top right of the figure, where COZ+ starts reading debugging symbols of the target application to construct a hash table that maps instructions to the corresponding source line. This hash table is essential to keep track of the application's threads at runtime. COZ+ constantly references this hash table to match the thread's program counter with a line that is selected for speedup. After processing the symbols and building a hash table, COZ+ creates a profiler handler and then executes the target application.

At runtime, whenever the application spawns a new thread (via `pthread_create()`), the profiler handler creates a new *sampler* and attaches it to the thread (also valid for the main thread). This is indicated by the purple boxes in the figure. Each sampler has a timer that interrupts the thread with a fixed frequency. Upon receiving a signal, the thread captures hardware/software counters (e.g. program counter and call stack) and saves them into an appropriate data structure. This is implemented via the *Linux perf events* [10] API which is a lightweight performance profiling tool in the Linux kernel. In order to control processing overhead, samples are processed in batches. COZ+ processes samples (*process samples* module in the figure) to determine if a thread is executing the line that is selected for speedup. If it is, COZ+ suspends other threads for a certain amount of time or might skip a thread if it is already in the wait state (e.g. acquiring a lock or I/O operation). The thread suspension is handled via a relatively complex mechanism that is shown as the *insert delay* module in the figure. Simply put, this module (1) calculates the amount of time each thread needs to suspend and (2) prevents inefficient thread-to-thread communication by orchestrating all suspensions through a global system. As indicated in the figure, we keep this module untouched in COZ+. Finally, when the application terminates, the profiler handler processes the data from the sampler's counters and reports the result. This is shown in the bottom left of Figure 6.

In the rest of this section, we discuss the different design and implementation deficiencies of COZ that limits its scalability and describe how COZ+ overcomes these limitations.

Optimizing symbol loading. Before the program begins, COZ records the executable debugging symbols (DWARF) of the program from linking format files (ELF) into a hash table. Reading and processing all the debugging information of Chromium with over 11 million lines of C/C++ code and almost 270K source files is impractical as it takes hours to read and allocate a large amount of memory at runtime. As a result, we only keep the *compilation units* that contain source files related to the rendering stages and prune the rest (shown by a dotted red box at the top of Figure 6).

To scan the Chromium source code for footprints of the rendering stages, we take advantage of Chromium traces [16]. Chromium traces record important browser activities including rendering

[4]COZ+ is available open source at https://gitlab.com/coz-plus/coz-plus.

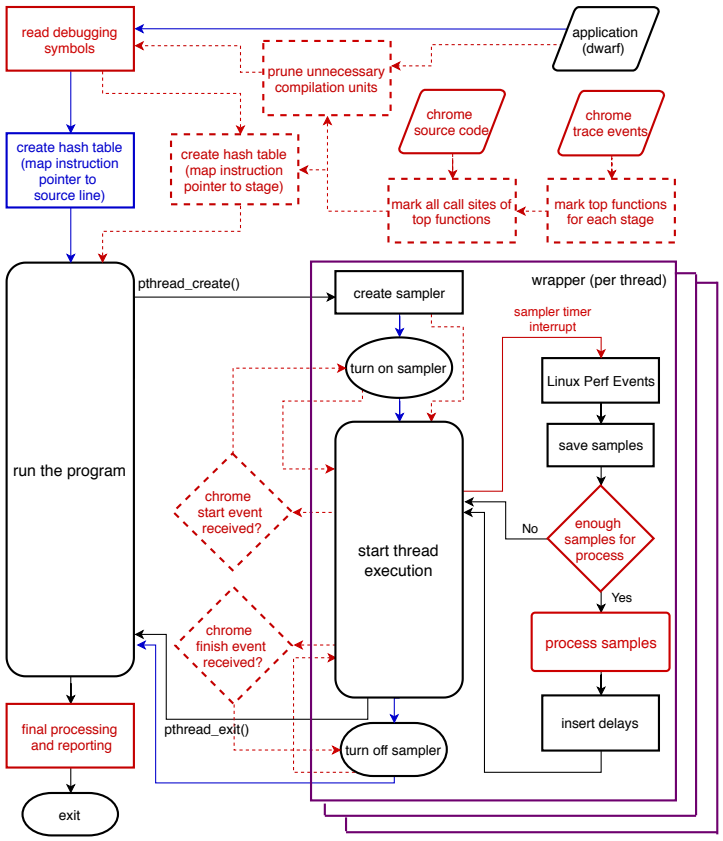


Fig. 6. COZ+ profiler architecture. Black arrows and boxes show the original COZ design. Solid red elements show our modifications and dotted red elements indicate new additions. Blue arrows and boxes show removed logic.

activities for profiling purposes. However, it is not necessary to include the source file for all of the low-level rendering activities in our case. It is sufficient to record debugging symbol of activity *a* and discard activity *b* if *a* encompasses *b* (*b* is always called inside *a*). For each stage, we select the set of non-overlapping top-activities that cover all stages. For example, Styling contains several non-overlapping top activities such as CSS tokenizing, CSS token parsing, updating DOM style, etc. Therefore, only the compilation units that contain top rendering activities are fed to COZ+ and the rest are pruned.

In addition, we observed that COZ symbol processing module for *compilation units* maps some of the debugging symbols to multiple lines. For example, COZ might map one inline symbol to different lines if the file containing the inline symbol is shared between different *compilation units*. We fix this issue in COZ+. More specifically, we first walk through DWARF file headers in *compilation units* and exclude unnecessary files (those that do not have top activities in our case) for line processing. With this optimization, the *total symbol processing time reduced from a couple of hours to less than a minute for each browser launch.*

By combining the output of the symbol processor and those locations that are highlighted as rendering top activities, we create a new hash table that maps debugging symbols to the

corresponding stages (shown by the red dotted box at the top left of the figure). This hash table enables the profiler to quickly determine the executing stage based on the instruction pointer throughout the execution. Compared to the hash table used in COZ, the COZ+ hash table is lighter (in terms of both access time and memory) as it only hashes the stage of the activities for relatively fewer symbols.

Flexible sampling. Sampling rate and batch size are two important factors that impact the performance and accuracy of causal profiling. The sampler frequency and batch size are hardcoded in COZ. In COZ+, we make these parameters configurable. For example, if the experiments are short (as in our case), then the sampling frequency should be high to capture all activities. Contrary to this, sampling with high frequency in applications with lengthy tasks does not have any advantage and increases the profiler overhead. Similarly, large batches, on the one hand, reduce sample processing overhead, but on the other hand, postpone the threads' suspension time which in turn sacrifices the accuracy. Furthermore, batch size and sampling frequency should be set by considering the number of active threads in the application. Since samples are processed asynchronously per thread but they influence all concurrent threads (in the thread suspension process), frequent sampling in applications with many threads greatly perturbs the application's normal execution path.

PLT takes only a few seconds and we observe that a large number of rendering activities take more than 20 milliseconds, therefore we set the sampling period to 2 milliseconds to have enough samples. Considering this sampling frequency and the number of active threads (usually around 40 threads), we estimate the optimal batch size range to be from 6-15. Batch sizes less than this range show pauses in the page rendering profile and those larger than this range shift the suspension time to more than 30 milliseconds per activity, which drops the accuracy significantly if the PLT is short. Therefore, for short web pages (PLT less than 4s), we set the batch size to 8 and for pages with longer PLT, we set the batch size to 10.

Sample processing adjustment. We modified the sample processing module (the red box titled *process samples* in Figure 6) in COZ+ for two reasons – (a) The original algorithm does not properly consider the sample's call sites. For example, COZ might wrongly accelerate a line if one of its call sites already exists in the symbol table even though neither that line nor any of its call sites match the selected line for speedup. (b) It is not compatible with our new symbol table. Algorithm 1 presents the pseudocode for COZ+ *process sample* module. For every unprocessed sample, COZ+ looks up the instruction pointer in the previously created hash table. If the symbol exists in the table, it checks its stage with the selected stage for speedup. If the stages match, the thread adds its local delay counter (which results in suspending other threads). When there is no symbol for the sample's instruction pointer, it is possible that the sample is captured in low-level activities. In this case, COZ+ walks through the sample's call sites and looks up every call site in the symbol table. As soon as it finds a relevant stage in the call stack, it adds local delay if they are a match. During processing sample's call sites, the procedure might find samples that belong to stages other than the selected stage. In this case, processing proceeds to the next sample without inserting any delay.

Multi-process profiling. Unfortunately, COZ could not profile multi-process applications. The profiler handler can only attach to the initial process and manages the samplers of the threads in the initial process. In our case, profiling the initial process (*browser process*) is not sufficient since almost all of the rendering activities reside in other processes (*renderer processes*). Therefore, we add multi-process profiling feature in COZ+ to make it compatible with most large applications. In our implementation, the profiler handler attaches to any process that is forked at runtime. Since each of the initiated processes has its own address space, they have to build a symbol table related to their loaded module addresses. Reading and processing these compilation units for every forked process at runtime is infeasible as it has significant overhead on the program. Therefore, COZ+ does symbol processing once at initialization and creates a symbol table with absolute addresses

Algorithm 1 Pseudo-code for *process samples* module in Figure 6

```
1:  procedure ProcessSamples
2:      samples[] ← get unprocessed samples
3:      n ← number of unprocessed samples
4:      selectedStage ← selected stage for speedup
5:      for i ← 1, n do
6:          ip ← GetInstructionPointer(samples[i])
7:          s ← FindStage(ip, hash)
8:          if s ≠ ∅ then                          ▷ symbol exists in hash table
9:              if s = selectedStage then
10:                 AddDelay()
11:             continue                           ▷ proceed to next sample
12:         callchain ← get call sites of samples[i]
13:         m ← length of callchain
14:         for j ← 1, m do
15:             s ← FindStage(callchain[j], hash)
16:             if s ≠ ∅ then                      ▷ symbol exists in hash table
17:                 if s = selectedStage then
18:                     AddDelay()
19:                 break                          ▷ proceed to next sample
```

within the compilation units in the shared memory. Whenever a new process is forked, it copies the symbol table from shared memory to its local memory and updates symbols with their respective address offsets.

Metrics and reporting: To achieve fairness between web pages, we use one metric representing PLT in all the measurements. This metric requires definite starting and ending locations. Therefore, in COZ+, we turn on and off samplers by the browser's events. The added module starts sampling when the `navigationStart` event is fired, which is the time the user enters the URL. However, developers can use other events such as `onBeforeRequest` (to start profiling when the first HTTP request is sent) or `onHeadersReceived` (to start sampling when the first byte is received) in this new implementation. The same procedure pauses threads' sampling. Since we are measuring PLT, we use the `loadFinish` event in our experiments[5]. Some developers may prefer the above-the-fold metric (the time that first content is shown on the screen), so they can use the *FP* (first paint) or *FMP* (first meaningful paint) events. Due to variability in page load time (e.g. fluctuation in network or browser garbage collection), COZ+ runs multiple experiments for each configuration. In order to save some time and space for our study (our study has around 12000 experiments), unnecessary data are eliminated from processing and reporting module.

3.3 Validation of COZ+

To verify the correctness of the integrated modules on top of COZ, we log all captured samples along with the timing report of the infused delays of all threads for 10 web pages. Then, we match them with the timing reports that come directly from the Google Developer Tool. For all the web pages, COZ+ was able to determine the executing stages 100% correctly. The amount of added

[5] A few studies use `DOMcontentLoad` (the time when all the HTML parsing is done and DOM is constructed) but our metric waits until all the DOM objects are loaded.

delay ($speedup \times sampling\,period \times number\,of\,matched\,samples$) shows less than 15% difference with calculated delay from theory.

In addition, we evaluate COZ+ to see how well it can predict the effect of optimization on the page loading process in a real scenario. Ideally, one should optimize the stages by a fixed amount and then compare the PLT of a test web page before and after this optimization. This approach is somewhat infeasible for the purpose of this paper since it requires significant research and development even for a small optimization in the current browsers. For this reason, we intuitively show this by bloating the browser code to simulate an unoptimized browser as our baseline.

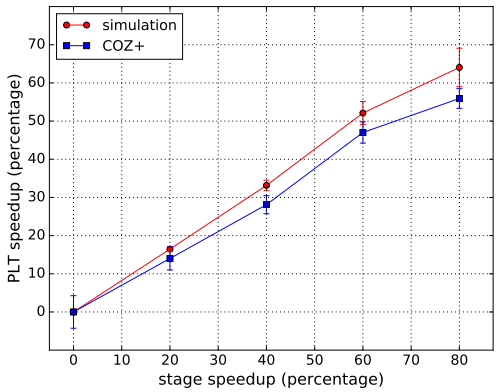


Fig. 7. Accuracy of what-if analysis with COZ+ on a test web page, `www.diply.com`.

As an example, we show our evaluations for the Scripting stage (since it turns out to be the most influential stage for optimization in section 5). We choose `www.diply.com` as a test web page because COZ+ estimated relatively large PLT improvement for this stage (approximately 26% PLT improvement for 80% JavaScript speedup). We modify the Chromium source code and slow down all Scripting activities such as script compiling, script executing, and callback functions invoked via events or time-outs in the loading of the test web page to 5× of its original time. The injected code keeps threads busy in CPU computations rather than holding the threads in the wait state as it might invalidate the integrity of the experiment in the presence of a job scheduler. Given the 5× extended version of the code as a baseline, it is possible to report PLT improvement after 80% and 20% stage optimizations (for the latter we compare the PLT with the 4x extended version of the code). Figure 7 compares the result from this simulation (red line) with the output of COZ+ on the baseline (blue line). As we can observe, COZ+ is able to accurately predict the impact of optimization and it shows less than 16% deviation from the simulation at 80% stage speedup and about 12% deviation at 20% stage speedup.

4 EXPERIMENTAL SETUP

System. We conduct all the experiments on a MacBook Air with 2.2 GHz Intel Core i7 processor (4 threads with hyperthreading) with 4 MB cache and 4 GB RAM. The host OS is 64-bit Ubuntu 16.04 LTS. Our second system has Intel Xeon E5-2630v3 2.4 GHz processor. This system has a total of 16 cores with 40 MB cache and 64 GB RAM hosting 64-bit CentOS 7.

Build setup. We use Chromium version 62.0.3167 and build it with Clang 3.8. We build COZ+ with the same compiler version and configuration. To evaluate the impact of key computation

activities in page loading, we build content-shell target of Chromium, which contains all the web platform features including HTML5 and GPU acceleration but excludes some of the Chrome-specific browsing features such as autofill, extension, and spellcheck. This makes our results more general to be used by other browsers, particularly other Webkit-based browsers. We include all the debugging symbols (`is-debug=true` and `symbol-level=2`) during the build as it is necessary for COZ+ to build the symbol table.

Configuration. We disable Chromium security Sandbox (`--no-sandbox`) because it runs Chromium in a protected environment and restricts COZ+ functionality on the browser. We also disable *Caching* in our experiments to observe the effect of the network on PLT. However, we repeat our experiments with caching enabled and demonstrate the effect of caching in section 5.2.

Experiment repeat. For each configuration, we load the page 10 times and report the median and average along with the variance.

Network. The system is connected to 100 Mbps Ethernet. To measure browser performance, we load web pages directly from the Internet, rather than using a local proxy. For wireless experiments, we use Wifi with 64 Mbps downlink speed. To emulate various network conditions, we use *Linux traffic control (tc)* [15] to limit bandwidth and network delay.

Web pages. Our test suite consists of top 100 web pages from Alexa Top 500 list in April 2018[1].

5 WHAT-IF ANALYSIS

In this section, we investigate the impact of the computation activities on PLT using COZ+. Then, we examine the effect of hardware, network connection, and browser caching on the behavior of computation activities and how they, in turn, impact PLT.

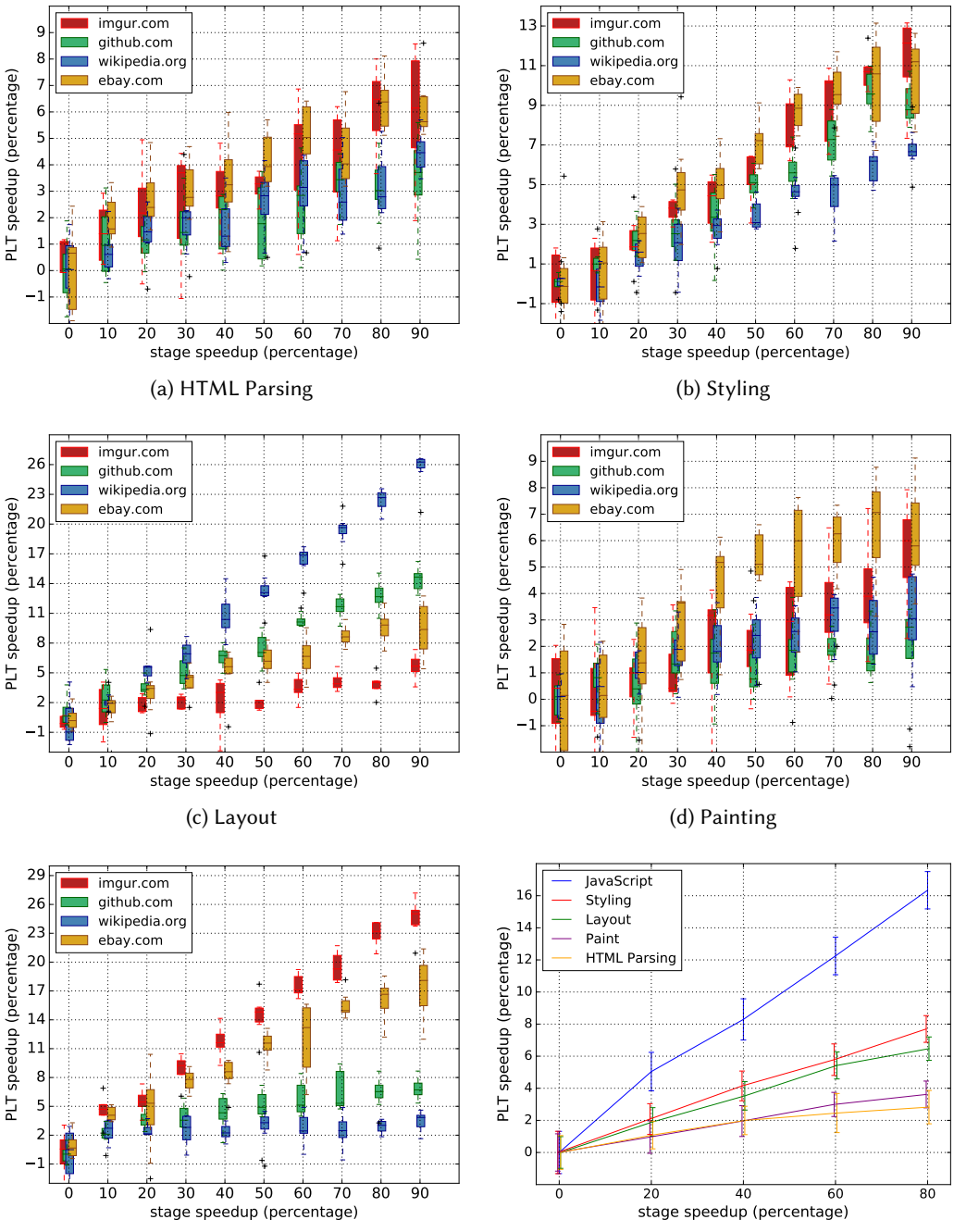
5.1 Impact of computation stages on PLT

We apply what-if analysis on all major page loading stages namely, HTML parsing, Styling, Layout, Scripting, and Painting (which includes compositing and layering in our measurements). To show the impact of these stages on browser performance, we run COZ+ to virtually accelerate activities in these stages and record the improved PLT. We gathered data for 10 evenly spaced speedups starting from 0% (no speedup) to 90% speedup (which means computing the stage 10× faster) for all stages. For each of the web pages in our test suite, we measure PLT 10 times for all pairs $(s, x) : \forall s \in$ stages, $\forall x \in$ speedups and calculate the average PLT with no speedup, $\overline{PLT}_{s,0}$. Then, we calculate PLT improvement for stage s and speedup x, $\Delta PLT_{s,x}$, as follows.

$$\Delta PLT_{s,x} = \frac{\overline{PLT}_{s,0} - PLT_{s,x}}{\overline{PLT}_{s,0}}$$

Plots [a-e] in Figure 8 illustrate the potential PLT improvement (PLT speedup) by stage as a function of speedup of that stage (stage speedup) for four popular web pages that exhibit different workload characteristics. Note that plots have different vertical scales. The plots also show the median and variability in the measurements. As we can observe, the benefit of stage improvement contributes to a diverse pattern among the web pages.

Finding 1. For the most part, we see a linear improvement in PLT. This indicates that there is not enough concurrency between stages during page load, otherwise, we expect to see a change in the slope of the graphs. However, in some cases, we can observe that different stage speedups have different impacts on web pages. For instance, in plot (e), if we optimize Scripting activities in `imgur.com` and `ebay.com` by 30%, COZ+ estimates an average page load performance improvement of about 8% and 9% respectively. However, if we can speed up this stage by 80%, `imgur.com` benefits 26% more than `ebay.com`.



(a) HTML Parsing (b) Styling

(c) Layout (d) Painting

(e) Scripting (f) Average PLT speedup

Fig. 8. (a-e) – Observed PLT improvement by accelerating browser stages namely, HTML parsing, Styling, Layout, Painting, and Scripting respectively for 4 popular example web pages. (f) – Average PLT improvement of Alexa Top 100 web pages. The boxplot displays the distribution of PLT speedup values. The boxes extend from the first to the third quartile (the 25th and 75th percentiles) with a line inside showing the median. Whiskers above and below the boxes extend from the minimum to the maximum value.

Finding 2. Web pages also show divergent patterns between stages. For example, `wikipedia.org` and `github.com` show marginal PLT improvement in Scripting in comparison to `imgur.com` and `ebay.com`. On the contrary, COZ+ estimates them to achieve significantly higher PLT improvements in Layout. This is related to the page content where one page could have many static elements and complex DOM that spends most of the time in HTML parsing and Layout while another page could have more dynamic elements to be evaluated by the JavaScript engine. Moreover, the organization of these elements can affect these patterns which are context-dependent.

Plots [a-e] show that the impact of stage optimization on PLT is content-dependent. However, to understand which of these stages is the primary bottleneck of browsers and furthermore, to predict the benefit of optimizing that stage, we calculate the average PLT improvement of Alexa top 100 web pages for each stage. Plot (f) in Figure 8 depicts PLT speedup as a function of stage speedup for the Chromium browser. The error bar shows the standard deviation of the mean.

Finding 3. We observe that JavaScript is the most influential stage compared to the other stages. This plot indicates that if developers can optimize JavaScript by 80%, they conceivably can improve browser page loading performance by almost 15%. Obviously, 80% improvement in any stage requires a significant amount of effort but even a 20% speedup of this stage can potentially reduce the average PLT by about 5% which can have a considerable impact on user experience, browser popularity, and web business revenue.

Multi-stage analysis. In some cases, an optimization might target multiple stages. Due to the inter-dependency between the stages, it is often difficult to estimate the final payoff based on individual stage payoffs. To address this, COZ+ supports multi-stage optimization with a distinct payoff per stage. For this purpose, COZ+ suspends concurrent threads whenever one of the stages from a list of given stages is executing. The amount of delay inserted is now proportional to the speedup of the executing stage. This feature aids developers in advanced decision making. One can now compare the benefit of two optimizations even if they do not target the same set of stages and/or have different speedups in similar stages.

Now, we repeat the what-if analysis by accelerating multiple influential stages simultaneously. The purple solid line in Figure 9 shows the projected PLT improvement when we accelerate the top two influential stages (JavaScript and Styling) simultaneously during page load. Here, 20% stage optimization refers to 20% speedup in both JavaScript and Styling stages. Although COZ+ allows distinct speedup values for different stages, we choose the same speedup to compare against single-stage analysis results. The dashed purple line is the sum of single-stage what-if speedups of JavaScript (blue line) and Styling (red line).

Finding 4. The overlap of these lines indicates that optimizing JavaScript and Styling has an additive payoff for the web pages in Chromium. We further track activities of these two stages and observe that a majority of these activities (which are co-dependent) execute on the same thread (i.e. main renderer thread) sequentially. While parts of script parsing run on other threads (web worker threads), it turns out there is no dependency between the former and Styling activities running on the main renderer thread.

Finding 5. We further extend our multi-stage analysis to include the third influential stage, Layout. In this case, we do observe a gap between optimizing all the three stages together compared to the sum of their individual speedups (black lines in Figure 9). This is due to dependencies between activities of these stages with activities on the I/O thread (i.e. network activities) that shift part of the critical path onto this thread.

5.2 Impact of PLT-variant factors

In this section, we examine the impact of key factors that influence PLT such as system architecture, network connection, and browser caching optimization on derived what-if graphs.

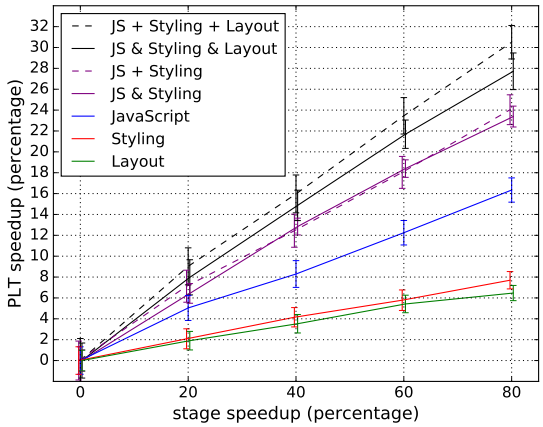


Fig. 9. Impact of accelerating multiple stages simultaneously on PLT. The solid lines correspond to accelerating single- or multi-stages using COZ+. The dotted lines are the sum of the individual stage speedups.

Evaluation on a different system. Given that system architecture influences computation activities, it is important to identify how much of the presented what-if results depend on the underlying hardware. Accordingly, we repeat the what-if analysis on our second machine (the system with Intel Xeon processor). Figure 10 shows the single-stage what-if analysis of Alexa top100 web pages.

Finding 6. Comparing this with the results from MacBook air (Figure 8(f)), we observe fairly similar trends for all the five stages (albeit higher variability in PLT). This implies that stage optimization payoff is fairly unrelated to the system architecture. Note that, while the impact of stages on PLT is consistent between the two systems, the web pages are loaded 20% faster on average on the second system.

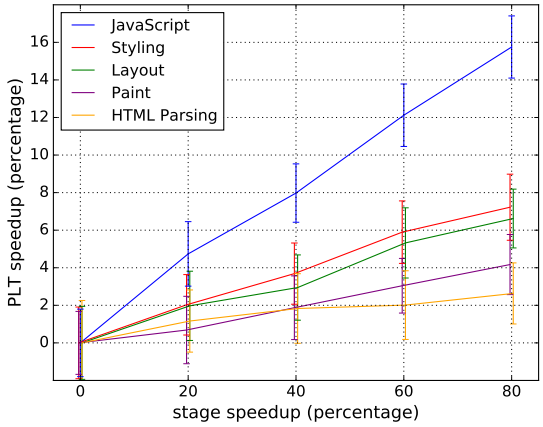


Fig. 10. Average PLT improvement of Alexa Top 100 web pages on a system with Intel Xeon E5-2630v3 processor.

Network connection. An interesting question that arises is: *Does the network have an impact on the outcome of the what-if analysis of computation stages?* In order to evaluate network effects on

potential optimization of the above stages, we conduct a similar experiment on the most influential stages from the previous analysis (namely, Scripting, Styling, and Layout) under different network conditions. We test different network connections, WiFi connection, and repeat the experiment on a smaller subset of the web pages (40 web pages randomly picked from our initial test suite). Network bandwidth and network delay are two factors that primarily influence resource loading and potentially the critical path. So, we emulate multiple network conditions by controlling these two network-dependent parameters.

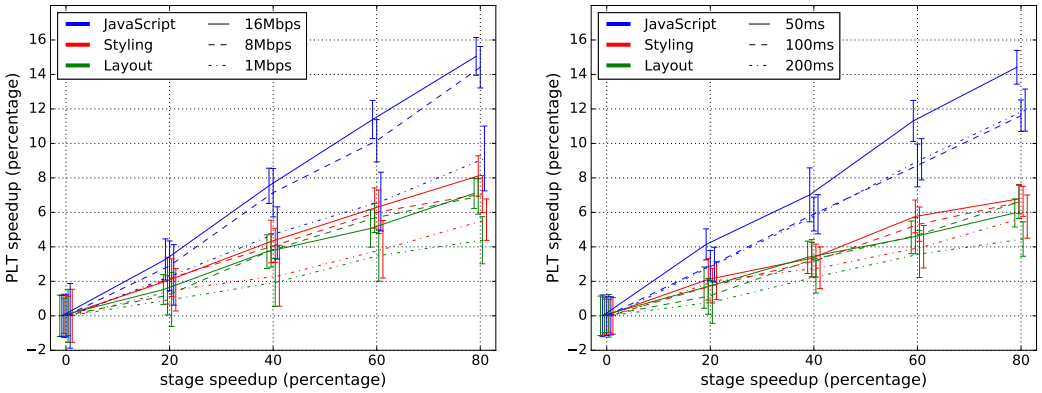


Fig. 11. Effect of varying network bandwidth (left) and network delay (right) on PLT speedup for the top 3 influential stages on 40 web pages of the test suite.

The left plot in Figure 11 shows how network bandwidth contributes to what-if analysis of the most influential stages. Different line styles are used to differentiate different network bandwidths, namely 1 Mbps, 8 Mbps, and 16 Mbps, and different colors are used for the 3 critical stages. Note that even though what-if graphs are shown with respect to the same baseline (i.e. 0 PLT improvement with no speedup), web pages have different baseline PLT under different network connections. For 16 Mbps, 8 Mbps, and 1 Mbps, the average PLT are 7.8, 8.9, and 12.6 seconds respectively.

Finding 7. An observation from this figure is that network bandwidth does not change the pattern of what-if plots and the order of the stages in terms of effectiveness. Scripting remains the most influential stage followed by Styling and Layout, respectively. Moreover, this figure indicates that improving network bandwidth increases the potential impact of computation stages on PLT which is not surprising since it likely increases the fraction of the computation stages on the critical path.

Finding 8. Network bandwidth has roughly the same contribution in the top three stages. For instance, the impact of Scripting on PLT is 1.7× more than Styling with 80% stage speedup under 1 Mbps bandwidth while this ratio remains almost constant (2×) at 8 Mbps bandwidth and (1.8×) at 16 Mbps. In general, stages' what-if graphs scale equivalently by varying the network bandwidths.

Finding 9. This plot also shows stages' what-if graphs exhibits a greater boost in PLT improvement (y-axis) by increasing network bandwidth from 1 to 8 Mbps in comparison to increasing bandwidth from 8 to 16 Mbps. For example, Scripting affects PLT roughly 60% more when bandwidth increases from 1 to 8 Mbps (by 7 Mbps), but only around 7% when it increases from 8 to 16 Mbps (by 8 Mbps). Contrasting this with the result from the previous experiment (100 Mbps connection), increasing the network bandwidth has an insignificant impact on what-if graph of stages on high-speed connections. This reflects that computation stages in the Chromium browser are mainly constrained by the computing power of the system and its dependency to other stages rather than downloading resources for networks with a bandwidth of about 8 Mbps and higher.

Results from different network delays are depicted in the right plot of Figure 11. We add 50 ms, 100 ms, and 200 ms delays to packets to increase the page RTT. The average PLT are 7.9 seconds (50 ms), 8.7 seconds (100 ms), and 10.4 seconds (200 ms).

Finding 10. As we can see, increasing the network delay diminishes the potential impact of the most influential stages. Even though we add 200 ms delay to web pages which is almost 5× the average RTT of our test suite, PLT speedup does not decrease significantly. For example, the PLT speedup drops by only 13% for 80% speedup in Styling.

Finding 11. Similar to the bandwidth experiment, network delay does not change the pattern of graphs meaning latency in fetching resources on the critical path is almost consistent between stages.

Caching. We enable caching and repeat the same experiment. The generated what-if graphs are almost identical for all the stages in comparison with caching disabled experiments since caching has a minor influence on PLT at 100 Mbps network connection (less than 5% for the majority of web pages in our test suite) indicating that computation activities are the bottleneck. So, we examine the caching effect under a slower network connection (1 Mbps). The average PLT without caching is 12.4 seconds and with caching is 8.0 seconds.

Finding 12. Figure 12 shows that PLT improvement drops significantly by disabling caching. As we can infer, an optimization targeting computation activities can approximately double its payoff by enabling caching at 1 Mbps network connection. Notably, the COZ+ what-if graphs with caching enabled reflect approximately similar stage impacts in comparison to stage impacts under high-speed connection. This is likely because almost all of the referenced objects before *LoadFinish* event are cached and retrieved quickly, so again computation activities build up the critical path.

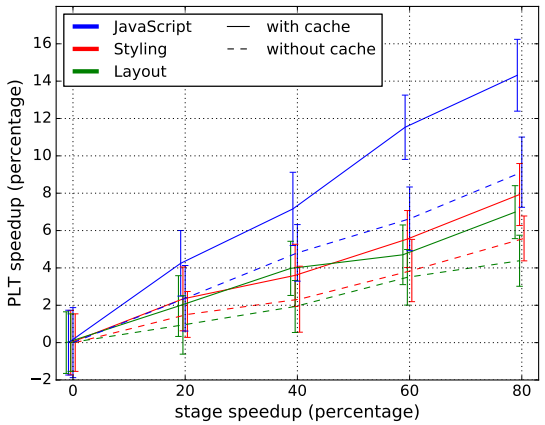


Fig. 12. Effect of caching on what-if graphs under a slow connection (1 Mbps).

6 RELATED WORK

Profiling and performance analysis tools. Majority of the browsers have their own profiler. *Gecko profiler* [8] for Mozilla Firefox and *Chrome profiler* [16] for Google Chrome are examples of such profilers. These profilers provide statistics about task timing, JavaScript call graph, memory usage, and network activities. Chrome takes a step further and collects a set of web-assistant tools under *Chrome DevTools* that guide web developers to diagnose their web pages [5]. The Chrome DevTools performance analyzer provides a brief summary of the time spent on each of the stages

and can also graphically show limited dependencies between fetched resources. However, they are not adequate for characterizing the behavior of the critical path and shift the what-if analysis to the user.

In addition to dedicated profilers, there are multiple tools that can assist users in recognizing the performance bottlenecks of web pages. PageSpeed Insights [4] is a web-tool that measures the *above-the-fold* load time and *full-page* load time for a given web page. Depending on the performance headroom of a web page load, it offers some suggestions (from a list of well-known web page optimizations such as "elimination of render-blocking JavaScript") on how that page can be improved. PageSpeed does not take into account network-dependent activities in performance analysis. Also, in the critical path exploration, it excludes lazy/eager binding dependencies and resource constraints as well as dependencies involving cached objects [40]. Yslow [22] is a similar tool that statically analyzes the page by crawling the DOM and capturing the information of DOM objects (size, whether it is gzipped, etc.). Then, it grades the page based on 23 pre-defined rules related to objects information and provides performance improvement suggestions. As far as we know, none of the existing tools are able to provide a quantitative and accurate what-if analysis as we offer for page loading.

Critical path analysis. WebProphet [33] reveals dependencies between objects via perturbation of network loads. It systematically delays individual object download time and builds *parental dependency graph (PDG)* for a web service. This framework can predict PLT based on PDG and client/server network conditions. Their *basic object timing* extractor is limited to network activities such as DNS lookup, establishing a TCP connection, and HTTP request/receive. It does not take into account the impact of computation activities in dependency extraction or in performance prediction.

The closest research to our measurement setup is Wprof [40], which is able to demystify page load performance. Wprof assigns a unique ID to the resources and individual loaded objects. It then derives a dependency graph from a set of pre-defined resource constraints and dependency policies between only those activities that are associated with loaded objects. Besides extracting the dependency graph, it breaks down the critical path of 150 web pages from computation and network aspects. Further examination of the computation activities (authors observe that computation activities make up 35% of the critical path), discloses HTML parsing costs more than Javascript and rendering stages in the critical path. This is in contrast to our findings that show that Javascript and Styling play a more critical role. Apart from dissimilarities in experimental setup [6], we believe the time breakdown of the critical path does not essentially manifest the impact of optimization since a component's optimization could affect the execution order of activities in an event-driven application.

Nejati et al. [37] extend Wprof for mobile devices and exploit the same methodology to compare non-mobile browser with mobile browser page load process. The key takeaway from the critical path breakdown is that computation activities outweigh network activities in the mobile browser contrary to the desktop browser, particularly for mobile websites. Even though they have analyzed page load critical path composition, similar to [40], it is debatable to derive what-if analysis based on static examination of the critical path. In addition to this limitation, [37] does not provide evaluations on rendering stages like painting or layout.

Prior to [37], Wang et al. studied the slowness of page loading on smartphones [42]. The authors use a fairly similar approach to Wprof to record the dependencies and timestamps of the main functions for IR operations (computation stages) as well as resource loading for 10 most visited

[6] [40] uses an older version of Chrome, v.22, which does not support some of the major page loading optimizations like Blink threaded HTML parser.

web pages. Despite the fact that they have tested on mobile devices with 3G and emulated Ethernet, their observations show significant divergence with our findings. As an example, with 32× speedup of Layout, they only observed 1.4% improvement in PLT which is in contradiction to our findings.

The advantage of causal profiling over previous approaches is that it eliminates dependency graph extraction, which in turn improves the reliability of measurements. This is crucial since existing tools do not take into account low-level inter-dependencies [40]. In addition, with causal profiling, it is possible to generate a quantitative what-if analysis of the page load considering the dynamic behavior of the critical path.

7 CONCLUSIONS

In this paper, we investigate and prioritize the bottleneck activities in modern web browsers. We primarily attempt to demonstrate the impact of these activities on the browser's page loading performance. To provide a meaningful estimation of how much benefit can be achieved by improving the critical activities, we present COZ+, a lightweight and customized profiling tool for current browsers. Incorporating COZ+ in the Chromium browser reveals that Scripting is the most influential stage for improving PLT for the Alexa top 100 most visited web pages. Our results show that improving this stage by 40% can potentially improve the performance of the Chromium browser by almost 8.5%. We also observe, contrary to some of the previous studies, that HTML parsing has a small contribution to PLT. Furthermore, our evaluation indicates that network conditions and caching influence the impact of computation activities. However, under typical network conditions (e.g. 8 Mbps connection), they have a negligible impact since the browser is bottlenecked by the computation activities. This would be of greater importance in mobile browsers since mobile devices have limited computing power. In our future work, we plan to extend COZ+ to mobile devices and analyze the mobile browser's limitations using a similar *what-if* style analysis. We believe COZ+ will be a useful tool and analysis technique for web researchers to prioritize their efforts on the most influential page load activities.

8 ACKNOWLEDGEMENTS

We thank Charlie Curtsinger, the developer of the original COZ profiler as well as Ben Greenstein and Zhen Wang at Google for their guidance and comments. We also thank the anonymous reviewers and our shepherd, Carey Williamson, for his careful reading and constructive feedbacks.

REFERENCES

[1] Alexa Top 500. http://www.alexa.com/topsites/countries/US.
[2] Google Developer: Analyzing Critical Rendering Path Performance. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp.
[3] Internet Explorer 9 Network Performance Improvements. https://blogs.msdn.microsoft.com/ie/2011/03/17/internet-explorer-9-network-performance-improvements/.
[4] PageSpeed Insights. https://developers.google.com/speed/pagespeed/insights/.
[5] Chrome DevTool. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.
[6] Chrome: Process Model. https://www.chromium.org/developers/design-documents/process-models.
[7] Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko.
[8] Gecko profiler. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.
[9] GNU gprof. https://sourceware.org/binutils/docs/gprof/.
[10] Linux Perf. http://man7.org/linux/man-pages/man1/perf.1.html.
[11] Microsoft Edge. https://docs.microsoft.com/en-us/microsoft-edge/.
[12] The Rendering Critical Path. https://www.chromium.org/developers/the-rendering-critical-path.
[13] SPDY. http://dev.chromium.org/spdy.
[14] SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.
[15] tc(8) - Linux man page. https://linux.die.net/man/8/tc.

[16] The Trace Event Profiling Tool. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool.
[17] Trident(MSHTML Reference). https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa741317(v=vs.85).
[18] V8. https://developers.google.com/v8/.
[19] The WebKit open source project. http://www.webkit.org.
[20] The most popular browsers. https://www.w3schools.com/browsers/.
[21] StatCounter Global Browser Stats. http://gs.statcounter.com/browser-market-sharemonthly-201802-201802-bar.
[22] YSlow. https://yslow.org.
[23] Blink. https://www.chromium.org/blink.
[24] COZ. https://github.com/plasma-umass/coz.
[25] Brian Anderson, Lars Bergstrom, Manish Goregaokar, Josh Matthews, Keegan McAllister, Jack Moffitt, and Simon Sapin. 2016. Engineering the servo web browser engine using Rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 81–89.
[26] Catalin-Alexandru Avram, Kenneth Salem, and Bernard Wong. 2014. Latency amplification: Characterizing the impact of web page content on load times. In *Reliable Distributed Systems Workshops (SRDSW), 2014 IEEE 33rd International Symposium on*. IEEE, 20–25.
[27] Carmen Badea, Mohammad R Haghighat, Alexandru Nicolau, and Alexander V Veidenbaum. 2010. Towards Parallelizing the Layout Engine of Firefox. In *Proc. of the 2nd USENIX Conf. on Hot topics in parallelism*. USENIX Assoc., 1–1.
[28] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. 2011. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM, 313–328.
[29] Calin Cascaval, Seth Fowler, Pablo Montesinos-Ortego, Wayne Piekarski, Mehrdad Reshadi, Behnam Robatmili, Michael Weber, and Vrajesh Bhavsar. 2013. ZOOMM: A Parallel Web Browser Engine for Multicore Mobile Devices. In *Proc. of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Prog. (PPoPP '13)*. ACM, New York, NY, USA, 271–280. https://doi.org/10.1145/2442516.2442543
[30] Charlie Curtsinger and Emery D Berger. 2015. COZ: Finding Code that Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 184–197.
[31] Darin Fisher, Brett Wilson, Ben Goodger, and Arnaud Weber. Multi-process browser architecture. US Patent 8,291,078.
[32] Christopher Grant Jones, Rose Liu, Leo Meyerovich, Krste Asanovic, and Rastislav Bodik. 2009. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*.
[33] Zhichun Li, Ming Zhang, Zhaosheng Zhu, Yan Chen, Albert G Greenberg, and Yi-Min Wang. 2010. WebProphet: Automating Performance Prediction for Web Services.. In *NSDI*, Vol. 10. 143–158.
[34] Leo A Meyerovich and Rastislav Bodik. 2010. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*. ACM, 711–720.
[35] Leo A. Meyerovich and Rastislav Bodik. 2010. Fast and Parallel Webpage Layout. In *Proceedings of the 19th International Conference on World Wide Web (WWW '10)*. ACM, New York, NY, USA, 711–720. http://doi.acm.org/10.1145/1772690.1772763
[36] James Mickens. 2010. Silo: Exploiting JavaScript and DOM Storage for Faster Page Loads.. In *WebApps*.
[37] Javad Nejati and Aruna Balasubramanian. 2016. An In-depth study of Mobile Browser Performance. In *Proc. of the 25th Intl. Conf. on WWW*. Intl. WWW Conf. Steering Committee, 1305–1315.
[38] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking.. In *NSDI*. 123–136.
[39] Sivasankar Radhakrishnan, Yuchung Cheng, Jerry Chu, Arvind Jain, and Barath Raghavan. 2011. TCP fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*. ACM, 21.
[40] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf.. In *NSDI*. 473–485.
[41] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian.. In *NSDI*. 109–122.
[42] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2011. Why Are Web Browsers Slow on Smartphones?. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile '11)*. ACM, New York, NY, USA.
[43] Zhen Wang, Felix Xiaozhu Lin, Lin Zhong, and Mansoor Chishtie. 2012. How far can client-only solutions go for mobile browser speed?. In *Proceedings of the 21st international conference on World Wide Web*. ACM, 31–40.
[44] Rohit Zambre, Lars Bergstrom, Laleh Aghababaie Beni, and Aparna Chandramowlishwaran. 2016. Parallel Performance-Energy Predictive Modeling of Browsers: Case Study of Servo. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*. IEEE, 22–31.

[45] Kaimin Zhang, Lu Wang, Aimin Pan, and Bin Benjamin Zhu. 2010. Smart caching for web browsers. In *Proceedings of the 19th international conference on World wide web*. ACM, 491–500.
[46] Ming Zhang, Yi-Min Wang, Albert Greenberg, and LI Zhichun. Web page load time prediction and simulation. US Patent 8,335,838.
[47] Zhijia Zhao, Michael Bebenita, Dave Herman, Jianhua Sun, and Xipeng Shen. 2013. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)* 10, 4 (2013), 44.

Received February 2019; revised March 2019; accepted April 2019