UNIVERSITY OF CALIFORNIA,
IRVINE

Adaptive Tools for Performance Analysis of Large-scale Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Behnam Pourghassemi

Dissertation Committee:
Associate Professor Aparna Chandramowlishwaran, Chair
Associate Professor Ardalan Amiri Sani
Assistant Professor Zhou Li

2021

DEDICATION

To my beloved parents and supportive brothers...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude to Aparna, my advisor and the committee chair, for guiding and supporting me over the years. She taught me how to conduct research, write decent papers, mentor juniors, and most significantly to believe in myself. I would like to thank her for everything she taught me, for every deadline that she stayed up late helping me, and for her forgiving "no worries Behnam" responses to my "sorry for being late" messages.

I am extremely grateful to have Professor Amiri Sani and Professor Li on my committee. I appreciate all of their constructive feedback and guidance throughout my Ph.D. Without their unwavering support, this dissertation would not have been possible.

I would like to extend my appreciation to faculty, researchers, and mentors with whom I have had the opportunity to collaborate, particularly, Professor Athina Markopolou (UC Irvine), Joo Hwan Lee (Samsung), Ben Greenstein (Google), and Abhinav Vishnu (AMD research). I also want to thank Professor Carey Williamson (University of Calgary) and Arif Merchant (Google research) for shepherding my papers in SIGMETRICS and Professor Charlie Curtsinger (Grinnell College) for developing Coz profiler and making it publicly available. Furthermore, I would like to appreciate all of the undergraduate and graduate students who helped me with my research, especially Jordan Bonecutter, Neil Thanawala, and Zhenghao Zhang.

Allow me to thank my lab-mates, Laleh, Rohit, Octavi, Frank, Mei, Bahareh and Ferran, for lending me a hand and all the good memories. My experience in the lab was substantially enhanced with their presence and support. I want to extend my appreciation to Farima who cherished my PhD journey and stood by me during the pandemic and the most stressful period of my life, as well as all of my fabulous friends, Ali, Sina, Pouya, Arash, Mehdi, Hessam, Rozhin, Sadjad, Nader, Shima, Saba, and Mohammad Javad. I know I can be difficult to deal with at the best of times but thank you all for being there for me when I needed it.

This dissertation was partially supported by the U.S. National Science Foundation under award number 1533917 and Google Faculty Research Award, so I appreciate NSF and Google for their generous support. I am also indebted to the University of California, Irvine for providing me with every opportunity and resource I needed to do my research and Association for Computing Machinery (ACM), particularly SIGMETRICS community, for nurturing and articulating my research.

Last but not least, I sincerely thank my family; My mother and my father for their unconditional love, support, and inspiration from the first day of my life; My elder brother, my best friend and my role model, for all of his sacrifices for my achievements, and my younger brother, who offered invaluable support and humor over the years as well as for all the sleeplessness he has endured because of my education.

VITA

Behnam Pourghassemi

EDUCATION

Doctor of Philosophy in Computer Engineering — **2021**
University of California, Irvine — *Irvine, California*

Master of Science in Computer Engineering — **2017**
University of California, Irvine — *Irvine, California*

Bachelor of Science in Electrical Engineering — **2015**
Sharif University of Technology — *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant — **2015–2021**
University of California, Irvine — *Irvine, California*

GPU Research Engineer Intern — **June 2018–Sep. 2018**
Samsung Semiconductor Inc. — *San Jose, California*

TEACHING EXPERIENCE

Teaching Assistant
Computer Networks (EECS 148) — **Spring 2021**
University of California, Irvine — *Irvine, California*

Engineering Data Structures & Algorithms (EECS 114) — **Winter 2019**
University of California, Irvine — *Irvine, California*

PUBLICATIONS

adPerf: Characterizing the Performance of Third-party Ads
Behnam Pourghassemi, Jordan Bonecutter, Zhou Li, and Aparna Chandramowlishwaran
ACM SIGMETRICS, 2021

Only Relative Speed Matters: Virtual Causal Profiling
Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran
ACM SIGMETRICS Performance Evaluation Review, 2020

On the Limits of Parallelizing Convolutional Neural Networks on GPUs
Behnam Pourghassemi, Chenghao Zhang, Joo Hwan Lee, and Aparna Chandramowlishwaran
ACM SPAA, 2020

Scalable Dynamic Analysis of Browsers for Privacy and Performance
Behnam Pourghassemi
ACM SIGMETRICS Performance Evaluation Review, 2019

What-if Analysis of Page Load Time in Web Browsers Using Causal Profiling
Behnam Pourghassemi, Ardalan Amiri Sani, and Aparna Chandramowlishwaran
ACM SIGMETRICS, 2019

cudacr: An In-kernel Application-level Checkpoint/Restart Scheme for CUDA-enabled GPUs
Behnam Pourghassemi, and Aparna Chandramowlishwaran
IEEE CLUSTER, 2017

Unsteady Navier-Stokes Computations on GPU Architectures
Bahareh Mostafazadeh, Ferran Marti, *Behnam Pourghassemi*, Fang Liu, and Aparna Chandramowlishwaran
AIAA Computational Fluid Dynamics Conference, 2017

PATENT

Platform for Concurrent Execution of GPU Operations
Behnam Pourghassemi, Joo Hwan Lee, and Yang Seok Ki
US Patent, Application Number 16/442,440

SOFTWARE

adPerf `http://gitlab.com/adPerf`
A framework for performance characterization of web ads

COZ+ `http://gitlab.com/coz-plus`
A high-performance causal profiler for Chrome browser

ABSTRACT OF THE DISSERTATION

Adaptive Tools for Performance Analysis of Large-scale Applications

By

Behnam Pourghassemi

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2021

Associate Professor Aparna Chandramowlishwaran, Chair

Performance is the critical feature in the design and productivity of software systems. A key to improving the performance of a program is a rigorous performance analysis (PA) followed by code optimization. Although the approach seems straightforward, it can be difficult in practice, especially when the application is large or has complex architecture and task dependencies. It becomes even more arduous when applications have parallel and non-deterministic executing models. There are also cases where the conventional PA will require exhaustive profiling runs or direct access to hardware. All of these hurdles limit the scope of PA and mislead optimization opportunities in front of developers.

In this dissertation, we focus on such applications of performance analysis and address challenges and critical questions revolving around them. The commonality among these applications is that conventional PA is either not applicable or provides insufficient guidance to researchers to incorporate optimal optimization. For example, traditional profilers cannot inform developers about the impact of optimization or expose the potentials of parallelization in large programs. Therefore, for every application of our study, we design and develop a novel performance analyzer that overcomes the shortcomings of existing solutions and exposes new opportunities for performance boosting. We leverage and scale the functionality of profilers in designing our performance analyzers. Overall, we implement four adaptive

tools for PA: (1) A high-performance causal profiler that characterizes page loading time and pinpoints critical spots for optimizing the performance of website and web browsers. (2) A virtual causal profiler that accelerates cross-platform software development by simulating the impact of optimizations on diverse systems. (3) A robust tool for performance analysis of online advertising which assists publishers to characterize the performance cost of web ads with high precision. (4) A profile-based performance analyzer that uncovers performance opportunities from parallelism in training deep learning models on GPUs and helps practitioners in determining the optimal configuration for concurrent execution of network layers.

All of our proposed PA techniques are meticulously designed to match the application constraints (e.g., parallel environment and massive codebase) and validated by large-scale experiments and measurements. We also provide our first-of-a-kind findings from our assessments in this dissertation.

Chapter 1

Introduction

Driven by the advances in the performance of computer hardware, software stacks are becoming larger, parallel, and more complex. Programs with multiple processes that spawn tens of threads are now ubiquitous in broad areas of computer science [86]. In high-performance computing, compute-intensive applications split the domain of simulation/calculation into a large number of tasks to be executed in parallel [114, 98, 113]. Similarly, with the advances in machine learning, larger models with multiple streams of execution are becoming state-of-the-art practice [139, 93]. Thus, ML practitioners exploit techniques such as concurrency and pipelining in implementing current models for reducing the training time [137]. Along with this trend in research, enterprise software systems and client-side applications are also driven by parallel and large codebase paradigms. A good example is modern web browsers such as Chrome [2] that has both multi-process architectures to isolate the tasks for security and preventing application breakage and a highly multi-threaded architecture to distribute rendering tasks among CPU cores.

A common concern around such complex and highly parallel environments is the difficulty in performance characterization, code optimization, and locating the root cause of performance

overhead [86]. Static code analysis, the de facto technique for code debugging, is broadly used for performance modeling and pinpointing performance bottlenecks. The pitfall of static analysis is in the applications that host diverse tasks with dense task dependency or in the applications where tasks are executed in a non-deterministic pattern [86, 91]. Alternatively, dynamic code analysis and runtime profiling have to be done for detailed performance modeling. As a result, performance engineers heavily rely on profilers. Although profilers show accurate statistics of timing, memory consumption, network communication, etc., they are still not powerful enough for performance analysis of parallel applications or to answer what-if questions. For instance, conventional profilers such as gprof and perf, may list time-consuming tasks (e.g., functions) of a program but do not interpret how much optimizing these tasks improve performance metrics [80, 155].

In this dissertation, we deal with large codebases and parallel applications and propose optimal solutions for performance modeling and workload characterization of such applications. Essentially, we leverage the scope of profilers and practically show how profile-based performance analysis can help developers with code optimization and shed light on a variety of performance-related questions. Overall, in this dissertation, we explore four application domains of performance analysis. For each, we propose a novel performance characterization technique that suits well with the application criteria. To realize proposed methods, for each application, we develop a new tool (i.e., performance analyzer) and conduct a series of performance measurements and validations and report our first-of-a-kind findings from assessments, guiding researchers on improving the performance of such applications. Here, we briefly introduce our four cases and discuss them in-depth later in dedicated chapters (Chapters 2 - 5).

1.1 Web Browser

Web browsers have become one of the most commonly used applications for desktop and mobile users. At a higher pace, web applications operating on browser engines are receiving widespread attention owing to their cross-platform support and simpler development process, wherein they need to have higher performance to compete with native applications. 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). 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 performance analysis has been conducted to inspect which parts of the browser's computation contribute to the performance overhead. To identify the source of overhead and provide answers to many performance "what-if" questions, a complete and quantitative analysis of the web browser's page loading process is required.

COZ+ and What-if Analysis

Given the parallel and large codebase of modern browsers, we apply an adaptive what-if analysis to precisely determine the impact of each computation stage such as HTML parsing and Layout on PLT. Unlike conventional profiling methods, causal profiling [80] (detailed in Chapter 2) can quantify the potential impact of optimizing a code segment on the program and has shown promising results in PA of parallel applications. To this end, we develop COZ+, a high-performance causal profiler capable of analyzing large software systems such as Chrome browser. COZ+ highlights the most influential spots for further optimization, which can be leveraged by browser developers and/or website designers. Using COZ+, we conduct what-if analysis over 100 most visited websites under different system configurations

and report our findings. For instance, COZ+ shows that optimizing JavaScript by 40% is expected to improve the Chrome desktop browser's page loading performance by more than 8.5% under average network connection.

1.2 Virtual Causal Analysis

A key application of causal profiling is to analyze what-if scenarios as discussed in the previous application. However, typical what-if analysis using causal profiling requires a large number of performance runs. Besides, the calculated performance models highly depend on the underlying machine resources, e.g., CPU, network, storage, so the impact of code optimization on one device does not translate directly to another. This is a major bottleneck in our ability to perform scalable performance analysis and greatly limits cross-platform software development. We address the above challenges by leveraging a unique property of causal profiling: *only relative performance of different resources affects the result of causal profiling, not their absolute performance.* We first analytically model and prove causal profiling, then, we assert the necessary condition to achieve virtual causal analysis on a secondary device.

VCOZ and Virtual Causal Profiling

Building upon the theory, we design VCoz, a virtual causal profiler that enables profiling applications on target devices using measurements on the host device. We implement a prototype of VCoz by tuning multiple hardware components to preserve the relative execution speeds of code segments. Our experiments on benchmarks that stress different system resources demonstrate that VCoz can generate causal profiling reports of Nexus 6P (an ARM-based device) on a host MacBook (x86 architecture) with less than 16% variance.

1.3 Online Advertising

Monetizing websites and web apps through online advertising is widespread in the web ecosystem, creating a billion-dollar market. This has led to the emergence of a vast network of tertiary ad providers and ad syndication to facilitate this growing market. In addition, the ability of today's browsers to load dynamic web pages with complex animations and Javascript has also transformed online advertising. Nowadays, online advertising forces publishers to integrate complicated ads from third-party domains. Besides privacy and security issues concerning this model, third-party web ads have a significant impact on webpage performance. The latter is a critical metric for optimization since it ultimately impacts user satisfaction. Unfortunately, there are limited literature studies on understanding the performance impacts of online advertising which we argue is as important as privacy and security.

To this end, we apply an in-depth and first-of-a-kind performance evaluation of web ads. We aim to characterize the cost by every component of an ad, so the publisher, ad syndicate, and advertiser can improve the ad's performance with detailed guidance. This is a challenging task given the complexity of contemporary web ads, the non-deterministic ways they are delivered (e.g., coming from real-time bidding system) and rendered (e.g., scheduled at runtime by the browser). Furthermore, there are no dedicated profilers that quantifies the performance cost of web ads and prior efforts rely primarily on adblockers that have several limitations.

adPerf and Performance Characterization

For the above assessment, we develop a tool, adPerf, for the Chrome browser that classifies page loading workloads into ad-related and main-content at the granularity of browser activities. adPerf leverages profiling traces for workload characterization and demystifies

performance overhead. Our evaluations with adPerf show that online advertising entails more than 15% of browser page loading workload and approximately 88% of that is spent on JavaScript. On smartphones, this additional cost of ads is 7% lower since mobile pages include fewer and well-optimized ads. We also track the sources and delivery chain of web ads and analyze performance considering the origin of the ad contents. We observe that 2 of the well-known third-party ad domains contribute to 35% of the ads performance cost and surprisingly, top news websites implicitly include unknown third-party ads which in some cases build up to more than 37% of the ads performance cost.

1.4 Deep Learning Models

Training a deep neural network (DNN) is a time-consuming process given the massive number of parameters that have to be learned, thus accelerating DNN training has been an area of significant research in the last couple of years. GPUs are currently the platform of choice for training neural networks and popular deep learning (DL) frameworks such as TensorFlow and PyTorch have GPU backends to execute DNN operations. While earlier networks such as AlexNet [152] and VGG [135] had a linear dependency between layers and operations, state-of-the-art networks such as ResNet [93], PathNet [88], and GoogleNet [139] have a non-linear structure that exhibits a higher level of inter-operation parallelism. This potential brings out several performance-related questions: is there any performance gain if operations are run in parallel on a single GPU? If so, which operations and their corresponding algorithms to choose and how to schedule them in parallel to maximize the performance gain?

catDog and Measurements

To answer these critical questions, we introduce a concurrency-aware tailor for DL operations on a GPU (aka. catDog). catDog leverages the profiling-based kernel characterization and makes a case for the need and potential benefit of exploiting parallelism in state-of-the-art non-linear networks. It also partitions GPU resources among DNN operations to accommodate concurrent execution on AMD GPUs. Based on our proposed profiling-based method, we identify tens of cases in popular models such as GoogleNet and ResNet where enabling concurrent layer execution on a GPU backend (e.g., in cuDNN library) can perform better than sequential kernel execution.

Chapter 2

What-if Analysis of Page Load Time

One of the notable applications wherein performance plays a critical role is website loading. Websites that load slowly have a lower rate of return, likewise, a web browser that doesn't deliver sufficient performance is likely to lose its market share. So, researchers and developers in the web community are constantly looking for an opportunity to optimize the performance. The way today's websites are loaded is quite sophisticated and requires a PA method that works with the parallel and complex architecture of modern web browsers. In this chapter, we utilize causal profiling and scale this technique to show how it assists us in answering many of the what-if questions raised in the web community.

2.1 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 *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 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 [4, 72, 148, 149]. Accordingly, various network infrastructure reconfiguration and client-side solutions have been proposed to diminish this source of overhead. Mitigating round-trip delay time, up-

[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

grading protocols along with the redesign of the browser's resource loading via prefetching, speculative loading, and smart caching are some of these techniques [149, 49, 159, 129].

On the other hand, more recent studies have implied that CPU-intensive tasks such as HTML parsing and DOM manipulation have a more significant contribution to the PLT [99, 111, 146, 116, 159]. Correspondingly, researchers have attempted to improve the performance of different stages in the page rendering pipeline [161, 112, 147]. Browser developers also parallelize compute-intensive stages of the browser and fine-tune concurrency to mitigate page loading slow-down [67, 74, 99, 111, 158].

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 [64, 48, 89]. Inter-dependency between these activities establishes a critical path in the rendering process, which is highly complex to analyze [146, 3, 66, 160]. 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 chapter, 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 [146, 116, 148]. 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 [117, 66].

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 [80], which indicates where 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 [61], the only implementation of the causal profiler (to the best of our knowledge). We integrate 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 Chrome browser [3] since it is currently the most popular browser for both desktop and mobile users [55][4]. Section 2.5 presents details of our implementation.

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 [12]. 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 [146, 116] 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% [116, 148]). In addition, we examine the impact of different factors such as hardware, caching optimization, and network connection (e.g., network bandwidth and network delay) on the behavior of computation activities. In Section 2.7 we outline all of our findings. Our findings shed

[3]to be exact, Chromium browser that is an open-source version of Chrome

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

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.

2.2 Background

2.2.1 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 unit behind web browsers is a *rendering engine* (a.k.a. layout engine), which transforms the web page plain content to the visual representation. Mozilla Firefox and Microsoft Internet Explorer (IE) have their own respective engines called *Gecko* [44] and *Trident* [52]. Microsoft's newer browser, Edge, uses *EdgeHTML* (a fork from Trident) [47]. The rest of the well-known browsers such as Google Chrome, Opera, and Safari are developed on top of the *Webkit* rendering engine [54].

Figure 2.1 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 maximize the overlap of network delay with processing of received chunks. Figure 2.2 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

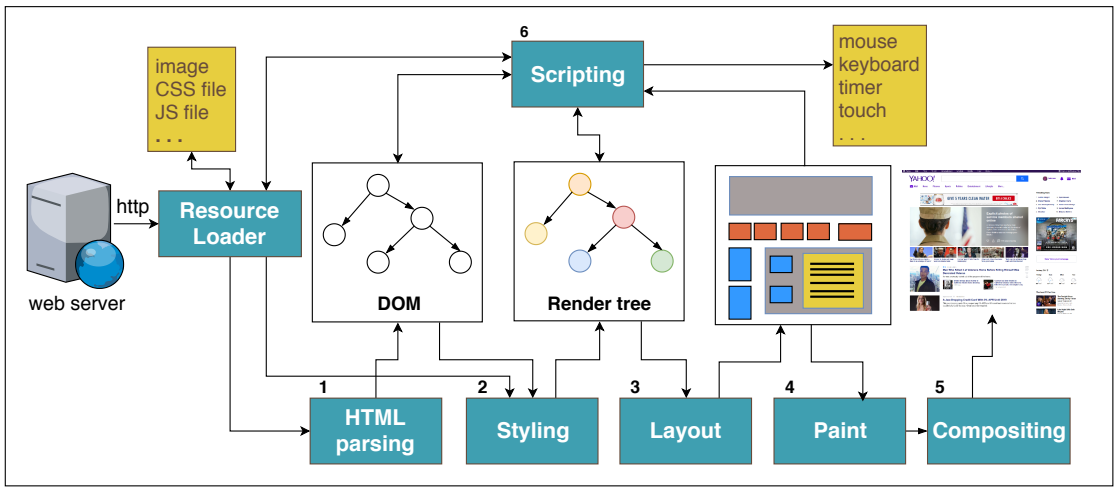


Figure 2.1: The general workflow for loading web pages.

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 (may) apply DNS lookup and stablish a TCP connection to download the object from the server or retrieve the object directly from the cache (Figure 2.2).

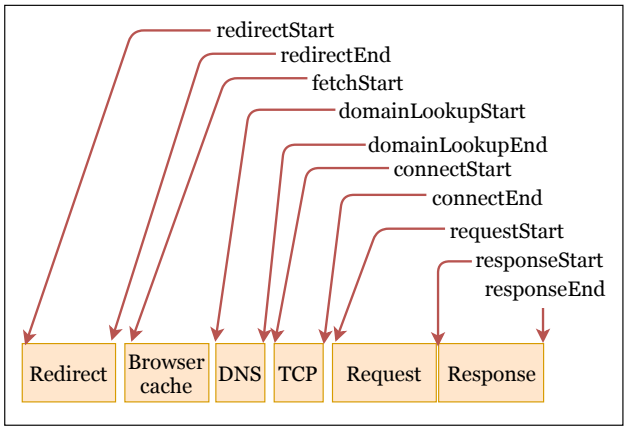


Figure 2.2: Resource loading stack.

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 for display. 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 element 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 [53] or *SpiderMonkey* in Mozilla Firefox [50]. 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 2.1.

2.2.2 Chrome Web Browser

According to StatCounter [56], Chrome is the most popular web browser in use for both desktop and mobile devices. As of May 2021, it has 64.7% of the browser's market share and no other browser comes close. More specifically, Apple Safari has the second place with 18.4% of the market share and Firefox lags far behind with only 3.4% of the market share.

Architecture. The rendering engine of Chrome, Blink [60], is forked from the popular Webkit engine [54]. Chrome exploits process-per-site-instance architecture to protect the overall browser from crashes, glitches, or malware in web pages [43]. In this architecture, the main process, *browser process* runs the UI and manages tabs. One *renderer process* is created per web page instance. Chrome 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

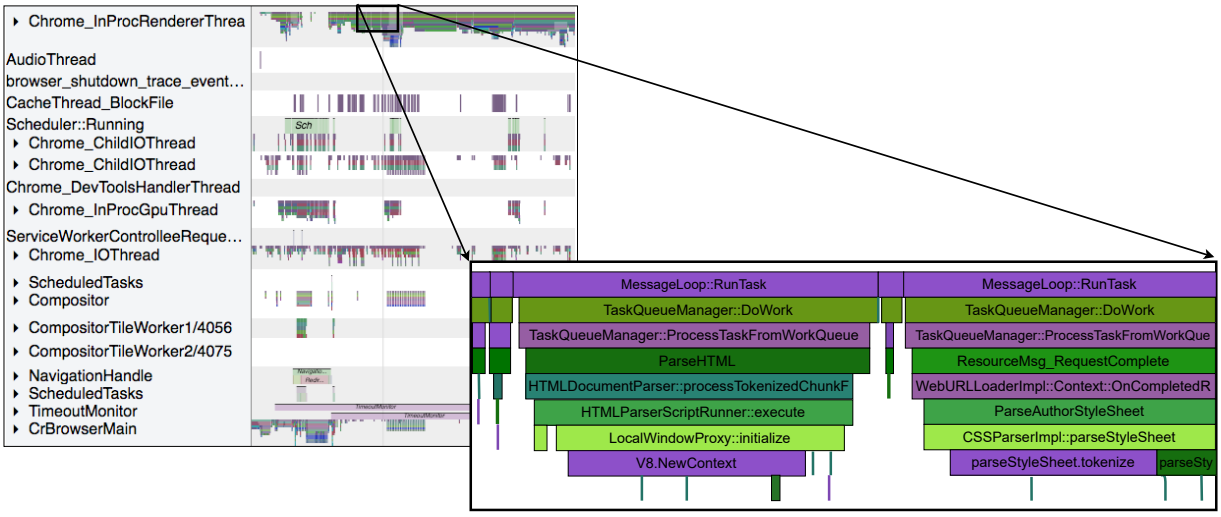


Figure 2.3: Page load time for `apple.com`. This timeline is obtained using the Chrome Trace Event Profiling Tool [51].

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. It may also 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 2.2 are managed by I/O threads [48]. Figure 2.3 shows a snapshot of the page loading timeline for `www.apple.com` obtained using the Chrome Trace Event Profiling Tool [51]. As we can see, multiple threads with different activities are involved in the page loading process.

2.3 Challenges in Critical Path Analysis

There exist inter-dependencies between browser stages during the page loading 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. [146] 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.

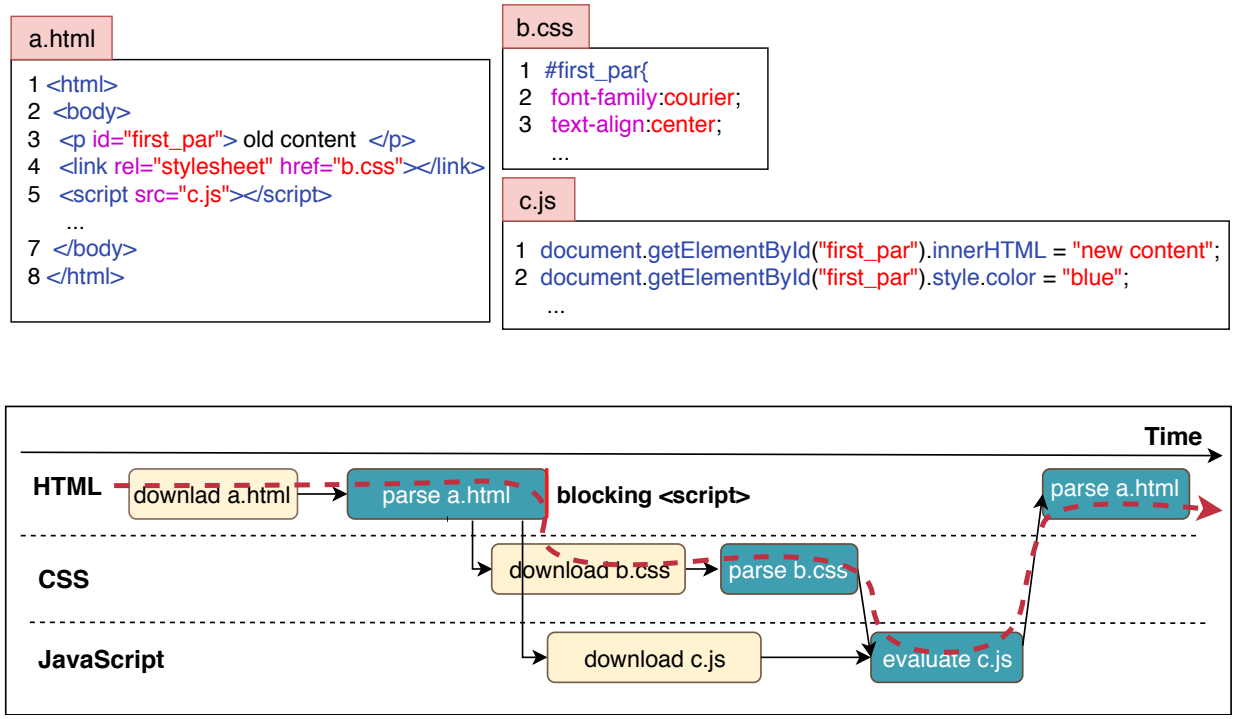


Figure 2.4: (Top) An example to illustrate the dependencies between the different activities. (Bottom) Timeline showing page load activities. Black arrows represent the dependencies between activities and the red dotted line shows the page load critical path.

Figure 2.4 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 2.4 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 analyzing the critical path and page loading bottlenecks extremely challenging [146, 3, 160]. Essentially, when we consider the large number of stage activities and multi-threaded executing paradigm of activities during page loading discussed in the previous section.

For *comprehensive what-if analysis* on modern web browsers with parallel and convoluted architecture, web researchers and browser developers have to use a suitable tool. Conventional profilers for browsers like the *Chrome profiler* in *Chrome developer tools* [42] 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* [101] 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 [80]. 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.

2.4 Causal Profiling

Causal profiling [80] is a novel method for finding performance bottlenecks and determining the impact of optimizations on a program. The Coz profiler [61] is the original implementation of the causal profiler. It is based on the idea of *virtual speedup* to find the impact of an optimization in a line of code (e.g., function call) on the total execution time of the program. In fact, virtual speed up simulates the behavior of code optimization by artificially slowing down other parts of the program.

Figure 2.5 illustrates the concept of *virtual speedup* [5] with a concrete example. The top timeline shows the execution of the original program with two threads running functions A, B, and C and the dependency between them. 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 20% (left) and 50% (right). 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. For the left timeline, it is 20% of the function A, and for the right, it is 50% of A. The difference between the execution time of the program after virtual speedup and the original time of the program with all inserted delays (indicated

[5] this is a different concept than virtual causal profiling that we propose in the next chapter

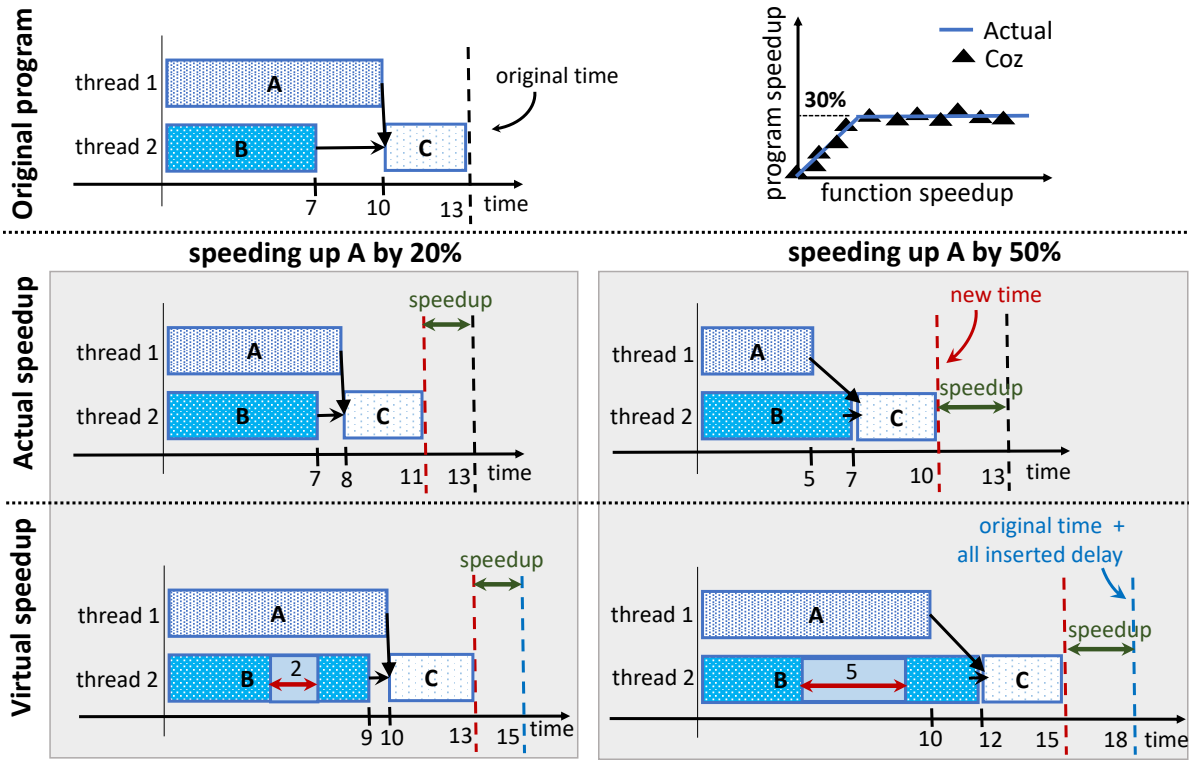


Figure 2.5: Illustration of the concept of *virtual speedup* and *causal profiling*.

by *speedup*) results in the same speedup as actually optimizing A (middle timeline). With Coz profiler, one can vary the amount of virtual speedup in A and plot the corresponding program speedup. The graph (also called what-if graph) for function A is shown in the top right corner of the figure. Ideally, this graph is expected to align with the graph generated from the actual optimization of function A.

2.5 COZ+: a High-performance Causal Profiler

A key contribution of this work is to *utilize causal profiling to apply what-if analysis on 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 current 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.

2.5.1 COZ+ Implementation

We originally intended to use Coz for what-if analysis of the Chrome 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 build COZ+, a comprehensive overhaul of Coz, which provides flexible profiling functionality for large applications as well as robust performance analysis capabilities for the Chrome browser. The original Coz profiler has approximately 4k LOC and we modify/add around **1k lines** to build COZ+[6]. 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 in future we will add support for other browsers.

Figure 2.6 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.

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

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* [46] 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 2.6.

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

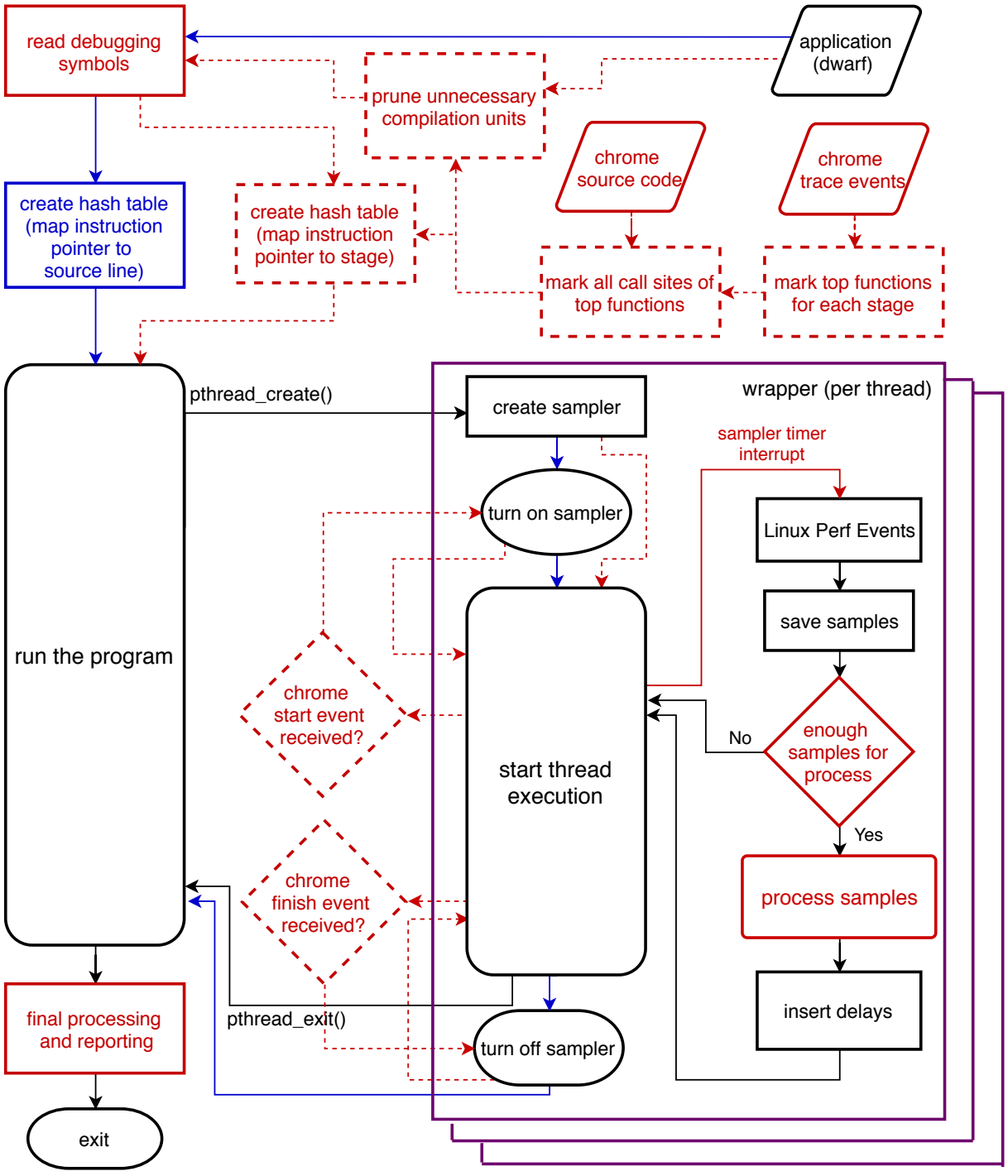


Figure 2.6: COZ+ profiler architecture. Black arrows and boxes show the original Coz design. Solid red elements show our modifications and dotted red elements indicate new additions. Blue arrows and boxes show removed logic.

(shown by a dotted red box at the top of Figure 2.6).

To scan the Chromium source code for footprints of the rendering stages, we take advantage of Chrome traces [51]. Chrome traces record important browser activities including rendering activities for profiling purposes [124]. 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 2.6) 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.

Algorithm 1 Pseudo-code for *process samples* module in Figure 2.6

```
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
```

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 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[7]. 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.

[7]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.

2.5.2 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 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 work 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.

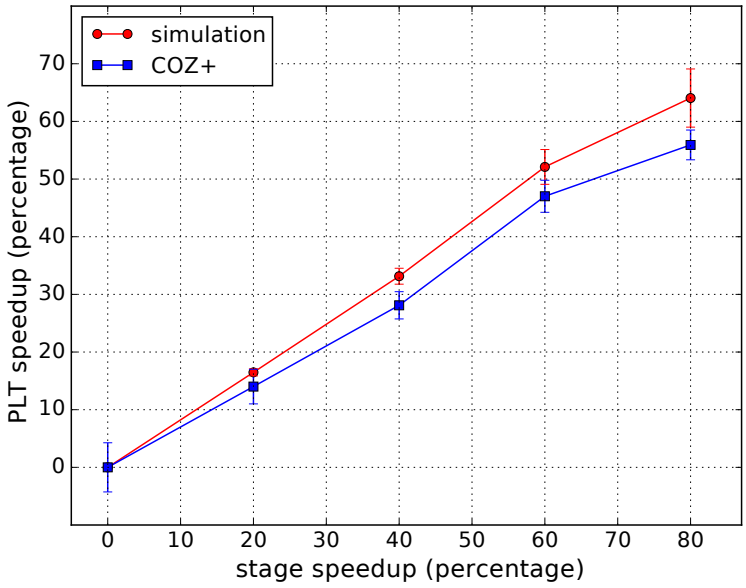


Figure 2.7: Accuracy of what-if analysis with COZ+ on a test web page, `www.diply.com`.

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 2.7). 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 Chrome 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 2.7 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.

2.6 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 Chrome security Sandbox (`-no-sandbox`) because it runs Chrome 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 2.7.2.

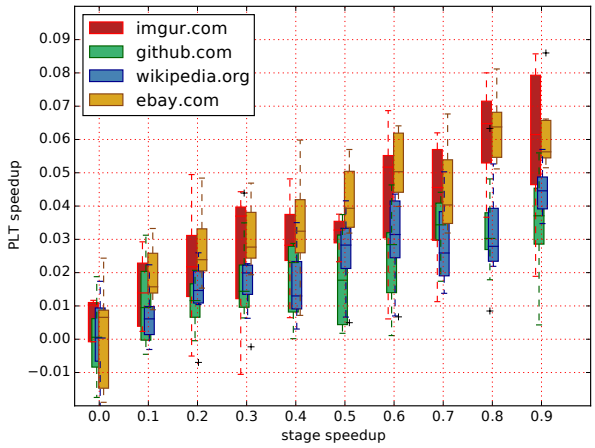
Experiment repeat. For each configuration, we load the page 10 times and report the median and average along with the variance.

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)* [30] to limit bandwidth and network delay.

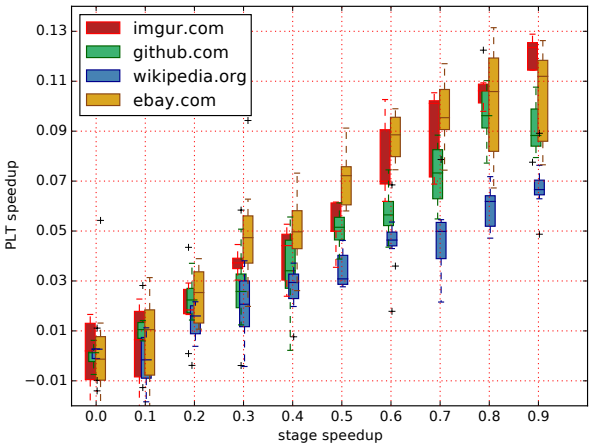
Web pages. Our test suite consists of top 100 web pages from Alexa Top 500 list in April 2018[12].

2.7 What-if Analysis

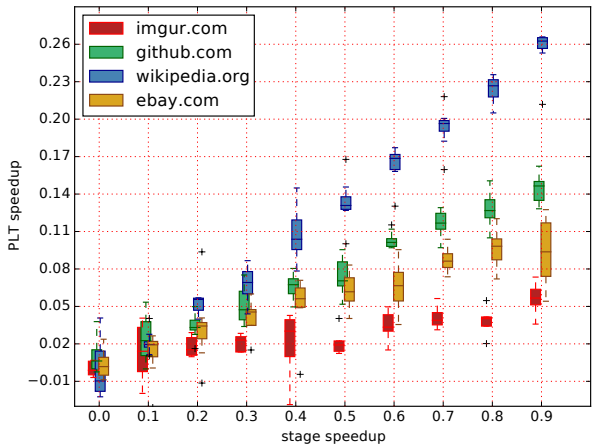
In this section, we investigate the impact of the computation activities on PLT using COZ+. Then, we examine the effect of hardware, network connection, and browser caching on the behavior of computation activities and how they, in turn, impact PLT.



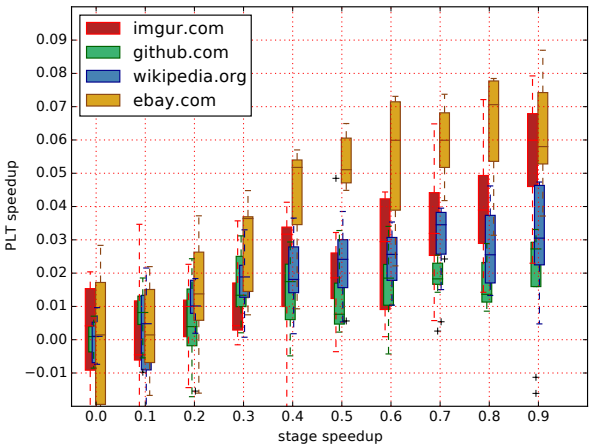
(a) HTML Parsing



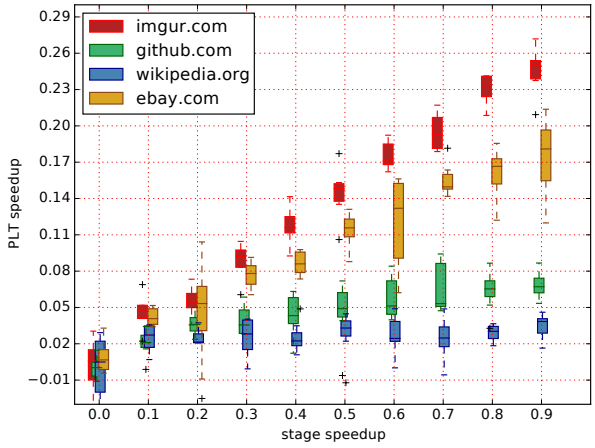
(b) Styling



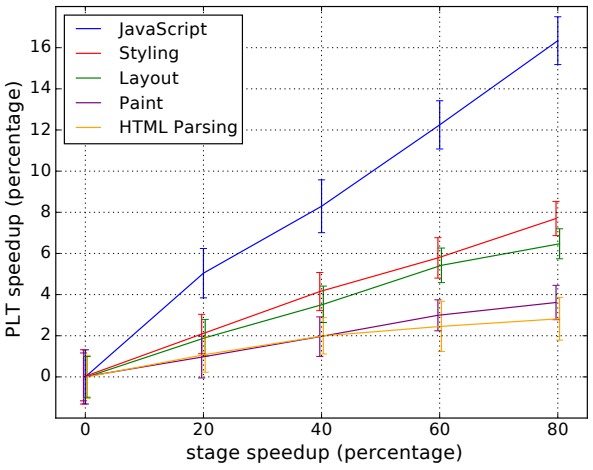
(c) Layout



(d) Painting



(e) Scripting



(f) Average PLT speedup

Figure 2.8: (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.

2.7.1 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 $(s, x) : \forall s \in \text{stages}, \forall x \in \text{speedups}$ 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 2.8 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`.

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 2.8 depicts PLT speedup as a function of stage speedup for the Chrome 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 2.9 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 Chrome. 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 2.9). 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.

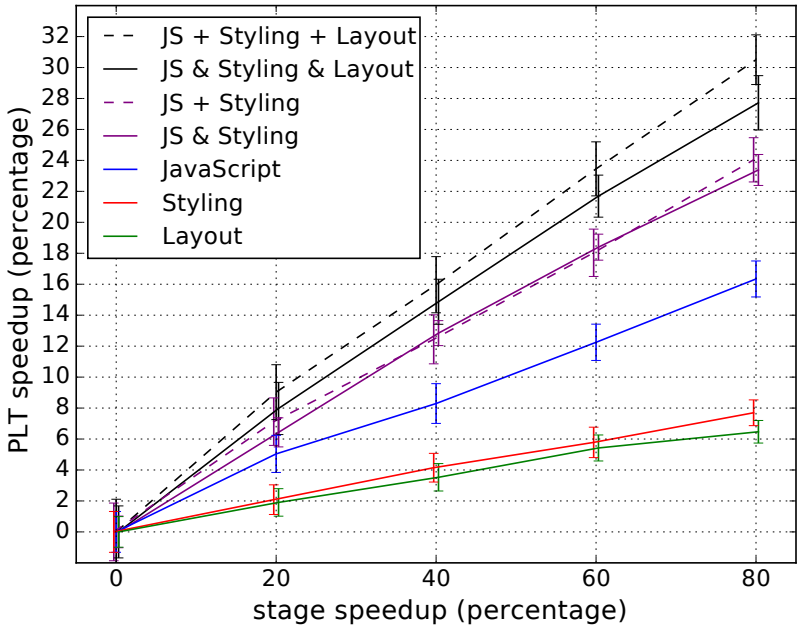


Figure 2.9: Impact of accelerating multiple stages simultaneously on PLT. The solid lines correspond to accelerating single- or multi-stages using COZ+. The dotted lines are the sum of the individual stage speedups.

2.7.2 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.

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 [127]. Accordingly, we repeat the what-if analysis on our second machine (the system with Intel Xeon processor). Figure 2.10 shows the single-stage what-if analysis of Alexa top100 web pages.

Finding 6. Comparing this with the results from MacBook air (Figure 2.8(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.

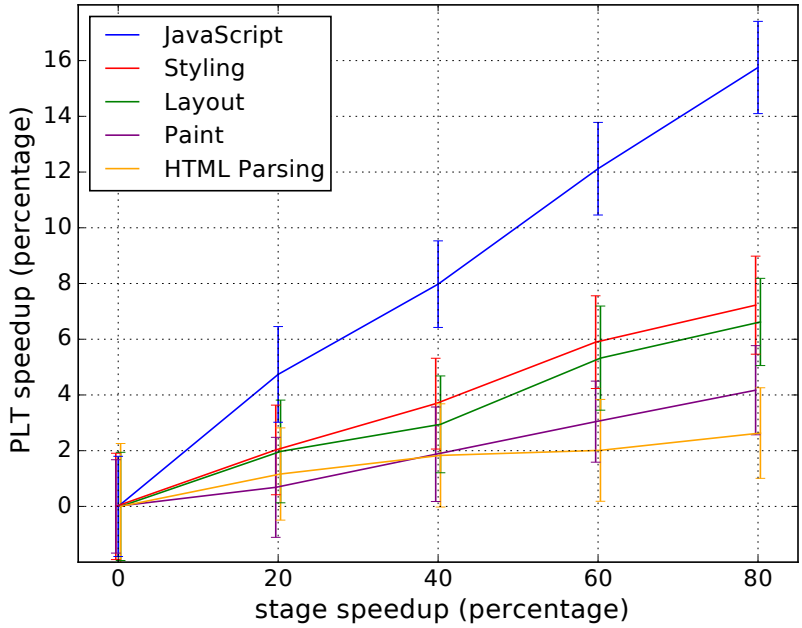


Figure 2.10: Average PLT improvement of Alexa Top 100 web pages on a system with Intel Xeon E5-2630v3 processor.

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.

The left plot in Figure 5.1 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

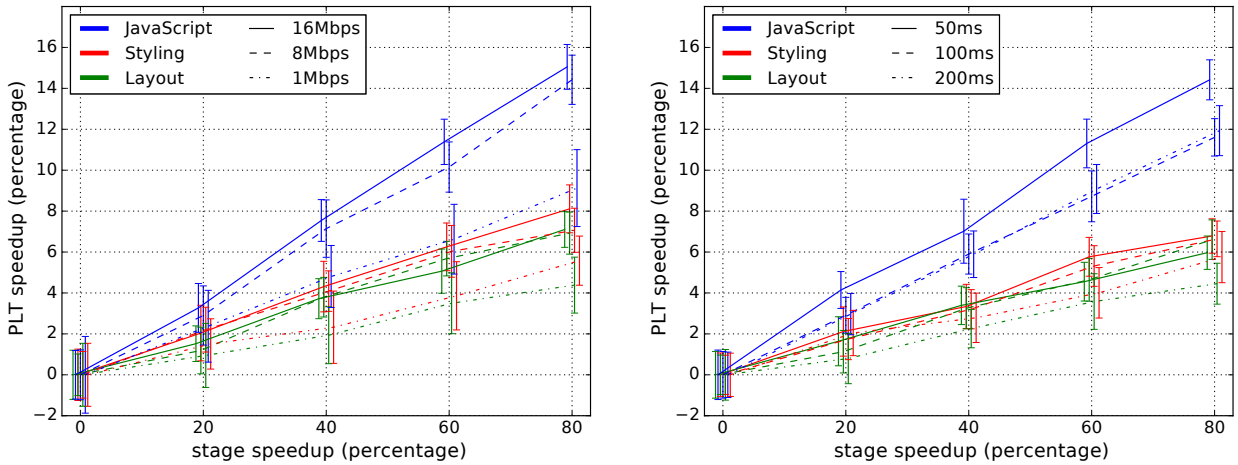


Figure 2.11: Effect of varying network bandwidth (left) and network delay (right) on PLT speedup for the top 3 influential stages on 40 web pages of the test suite.

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 Chrome 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 5.1. 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 2.12 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.

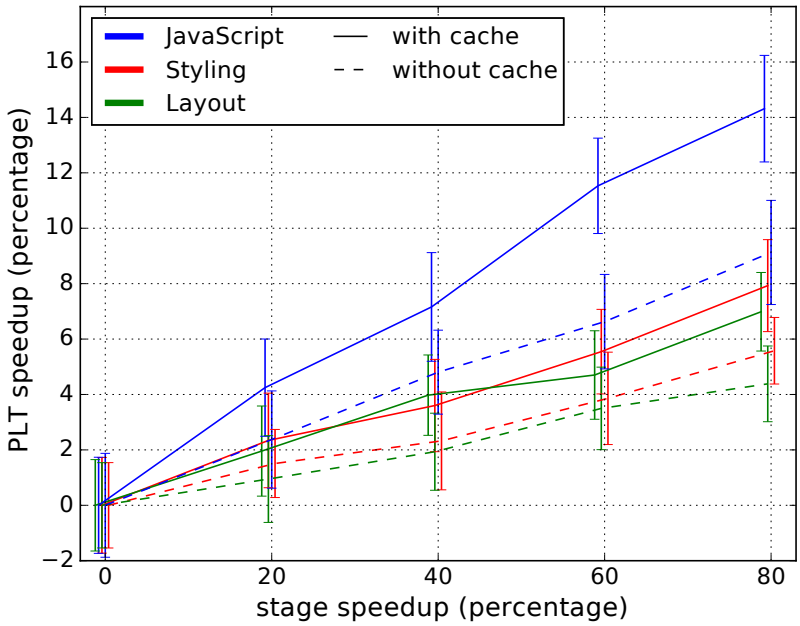


Figure 2.12: Effect of caching on what-if graphs under a slow connection (1 Mbps).

2.8 Related Work

Profiling and performance analysis tools. Majority of the browsers have their own profiler. *Gecko profiler* [45] for Mozilla Firefox and *Chrome profiler* [51] 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 [42]. 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 [41] 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 [146]. Yslow [59] 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 [110] 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 [146], 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 [8], 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. [116] 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 [146], it is debatable to derive what-if analysis based on static examination of the critical path. In addition to this limitation, [116] does not provide evaluations on rendering stages like painting or layout.

Prior to [116], Wang et al. studied the slowness of page loading on smartphones [148]. 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 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.

[8] [146] uses an older version of Chrome, v.22, which does not support some of the major page loading optimizations like Blink threaded HTML parser.

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 [146]. 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.

2.9 Conclusions

In this chapter, 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 Chrome 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 next work, we 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.

Chapter 3

Virtual Causal Profiling

Causal profiling has shown promising results in performance analysis. As we observed in the previous chapter, it is a powerful technique for what-if scenarios and pinpointing optimization opportunities in large and parallel applications such as web browsers. However, in some cases, for instance cross-platform application development, the developer needs to test the outcome of an optimization on various target platforms before integrating it into the product. Therefore, the PA has to cover a wide range of systems and configurations in a short amount of time. Unfortunately, the current state of causal profiling cannot accommodate this demand. In this chapter, we leverage our profile-based technique for what-if analysis and introduce virtual causal profiling. Similar to the concept of virtual machines, the virtual causal profiler performs performance runs in a virtual environment that mimics the behavior of the target system. Developers can use virtual causal profiling to run performance tests for a variety of systems and settings on a single machine (like multiple instances of VMs on one host). They can also use any system they own as a host, allowing them to divide the profiling workload over all available machines (multi-host), resulting in a faster PA time.

3.1 Introduction

Profiling tools are fundamental to the system design and implementation process. They serve several functions such as pinpointing bottlenecks, guiding optimizations, and pruning design space in the overall goal to improve system performance. *Causal profiling* is one such powerful profiling technique that has been successful in analyzing the performance bottlenecks of large and complex software systems such as the Chrome browser [123] and SQLite [80].

However, profiling can also be time-consuming and pose limits on cross-platform application optimization. Performance analysis on a diverse set of systems and configurations requires access to a broad range of devices. Such a setup is challenging to achieve in an academic research lab which restricts researchers to scope their profiling to a small number of systems.

This problem seems to be fundamental. That is, it seems like the only way to increase the coverage is to purchase and use a larger number of devices. Using Virtual Machines (VMs) in public cloud infrastructures, such as Amazon Elastic Compute Cloud (EC2) does not seem to be a feasible solution as the hardware specification of a VM can be different from that of the actual device, such as smartphones, tablets, desktops, laptops, and Chromebooks. This dissimilarity can affect the result of profiling to an extent where the conclusions cannot be relied upon. Cycle-accurate simulators and full-system emulators such as gem5 [70], QEMU [69], and Android Studio emulator [13] provide more accurate timing and performance characterization of applications. Although promising, instruction set simulators are prohibitively slow for full software stack performance analysis [138, 90, 143] and infeasible for *what-if* analyses under different scenarios that require a large number of experiments [80, 123]. Besides, conventional profiling tools typically do not work on top of such simulators [143].

In this chapter, we argue that while this problem is fundamental in the general case, we show that causal profiling provides a *unique opportunity* to address this using virtual performance

analysis. More specifically, we make a fundamental observation: *the result of causal profiling only depends on the relative execution speed of the application code segments*. Indeed, this relativity is at the core of causal profiling as it measures the potential speedup contributed by a program segment by slowing down all other code segments. Therefore, we can emulate the hardware configuration of, say, an ARM smartphone on an x86 laptop for causal profiling by controlling the relative performance of various resources such as CPU, network, and storage.

In this chapter:

1. We present an analytical model and proof of concept for causal profiling, a notable missing piece in the original paper. Then, we prove a necessary condition for virtual causal profiling on a secondary device.

2. Building on the above theory, we design VCoz, an infrastructure for virtual causal profiling that measures the impact of program speedups on various devices through hardware tunning of the host system. We implement a prototype of VCoz and port the causal profiler (originally designed for x86 architectures) to mobile devices (based on ARM) to validate our theory and prototype.

3. We test VCoz's cross-platform application optimization capabilities on multiple benchmarks with different workloads with MacBook Air as the host device and Nexus 6P as the target device. The experiments demonstrate that VCoz can predict the result of causal profiling with less than 16% variance while the original Coz profiler [61] misses the optimization opportunities.

3.2 Virtual COZ

Even though Coz works in practice and the concept of causal profiling is comprehensible by examples, the authors do not provide a proof of their method in the original paper [80].

Therefore, in this section, we first prove the concept of virtual speedup and causal profiling with a mathematical paradigm. Then, we extract the critical condition for soundness of causal profiling which is the retention of the relative speed of code segments. Following that, we describe our methodology to translate this theory to practice.

3.2.1 Theory and Mathematical Formulation

Suppose we have a program that runs on N threads. We can then divide the program into smaller code segments such that each segment (e.g. a function or basic block) runs entirely on only one thread. However, there might be dependencies between code segments, creating a critical path that determines the execution time of the program.

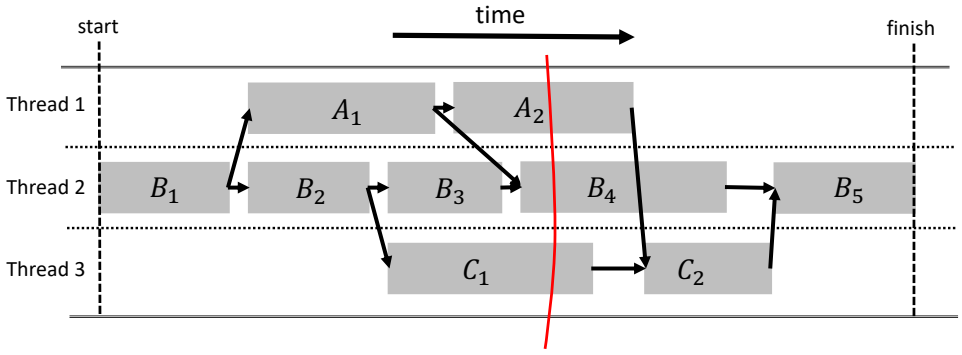


Figure 3.1: An example timeline for a program running on 3 threads. Edges show dependency between code segments in the program.

Figure 3.1 illustrates an example program with 3 threads and 9 segments and the dependencies between them. Generally, for any code segment, s, we define $S(s)$ and $F(s)$ as the start and end times of s, $E(s) = F(s) - S(s)$ is the execution time of s, and $D(s)$ is the set of segments that s depends on. For any s, $F(s) = E(s) + \max(F(D(s))$. Therefore, the total execution time, T is given by, $T = E(b_5) + \max(F(b_4), F(c_2))$ for the program in Figure 3.1. Without loss of generality, we present the proof for the program in the figure for readability. However, it applies to any program with multiple threads and arbitrary dependencies

between code segments.

Recursively expanding the above equation and assuming the initial segment starts at time 0 ($S(b_1) = 0$), we have,

$$T = E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} \tag{3.1}$$

Now, let us speed up an arbitrary segment s by ε seconds. The new execution time, T_{new}, can be calculated from the above equation by updating $E(s)$ (i.e., subtract ε). We state that T_{new} is equal to $T_{virtual} - \varepsilon$ where $T_{virtual}$ is defined as the total time derived by adding ε to all the segments in Equation 3.1 that a cutset (red line in Figure 3.1) passes through except the selected segment for speedup. In our example, if we desire to speedup segment a_2, we add ε to $E(b_4)$ and $E(c_1)$ but keep $E(a_2)$ unchanged to calculate $T_{virtual}$.

$$
\begin{aligned}
T_{virtual} &= E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) + \varepsilon \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) + \varepsilon \\ E(b_1) + E(b_2) + E(c_1) + \varepsilon + E(c_2) \end{pmatrix} \\
&= E(b_5) + \max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) - \varepsilon + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} + \varepsilon \\
&= T_{new} + \varepsilon
\end{aligned} \tag{3.2}
$$

Using 3.2, program speedup, S, relates to $T_{virtual}$ by,

$$
S = \frac{T - T_{new}}{T} = \frac{T - (T_{virtual} - \varepsilon)}{T} = \frac{(T + \varepsilon) - T_{virtual}}{T} \tag{3.3}
$$

Figure 2 illustrates the above mathematical formulation of speedup that underlines causal profiling [80]. Given this definition of speedup, the theorem that lays the foundation for the proposed virtual infrastructure is as follows.

Theorem. *If the execution time of all the segments is scaled by a constant factor α (i.e. $E(s^*) = \alpha E(s)$) and the speedup in a selected segment is also scaled by the same factor (i.e. $\varepsilon^* = \alpha\varepsilon$), then the new program speedup, S^* is the same as S and given by $\frac{(T + \varepsilon) - T_{virtual}}{T}$.*

Proof. *We can prove the above theorem by extending Equation 3.1. Since the execution time of all the segments is scaled by α, the new execution time of the program, T^*, is now given by,*

$$T^* = \alpha E(b_5) + \alpha.\max \begin{pmatrix} E(b_1) + E(a_1) + E(a_2) + E(c_2) \\ E(b_1) + E(a_1) + E(b_4) \\ E(b_1) + E(b_2) + E(b_3) + E(b_4) \\ E(b_1) + E(b_2) + E(c_1) + E(c_2) \end{pmatrix} = \alpha T \tag{3.4}$$

Similarly, we can derive $T^*_{new} = \alpha T_{new}$ and $T^*_{virtual} = \alpha T_{virtual}$ (omitted due to space constraints). Note that ε is also scaled by α. Finally, combining Equations 3.3 and 3.4:

$$S^* = \frac{(T^* + \alpha\varepsilon) - T^*_{virtual}}{T^*} = \frac{(T + \varepsilon) - T_{virtual}}{T} \tag{3.5}$$

3.2.2 VCoz: Theory to Practice

The proved theorem is important and functional as it asserts that casual profiling of an application on one system will be similar to the causal profiling of that application on another system if all the code segments run x% faster or slower. In practice, every code segment is built from a series of CPU, memory, and I/O operations. So, we can hypothetically split a code segment into smaller slices of only CPU or memory or I/O operations as demonstrated in figure 3.2 by different colors. In figure 3.2(a) two example code segments (A and B) are run on a target device. The relative execution time of these two code segments is

78/56 = 1.4. Figure 3.2(b) shows the same two segments executed on the host device with dissimilar hardware components. In this example, the host has a processor that runs 20% slower ($\alpha_{cpu} = 1.2$), a memory with 50% slower bandwidth ($\alpha_{mem} = 1.5$), and I/O systems (including disk, network, etc.) that operate 2× faster ($\alpha_{io} = 0.5$). Therefore, the relative execution time of the code segments is different on the host (84/71 = 1.2), leading to incompatible causal profiling of the target device. To achieve a similar relation, the different slices in the code segments have to retain an identical scaling factor from target to host.

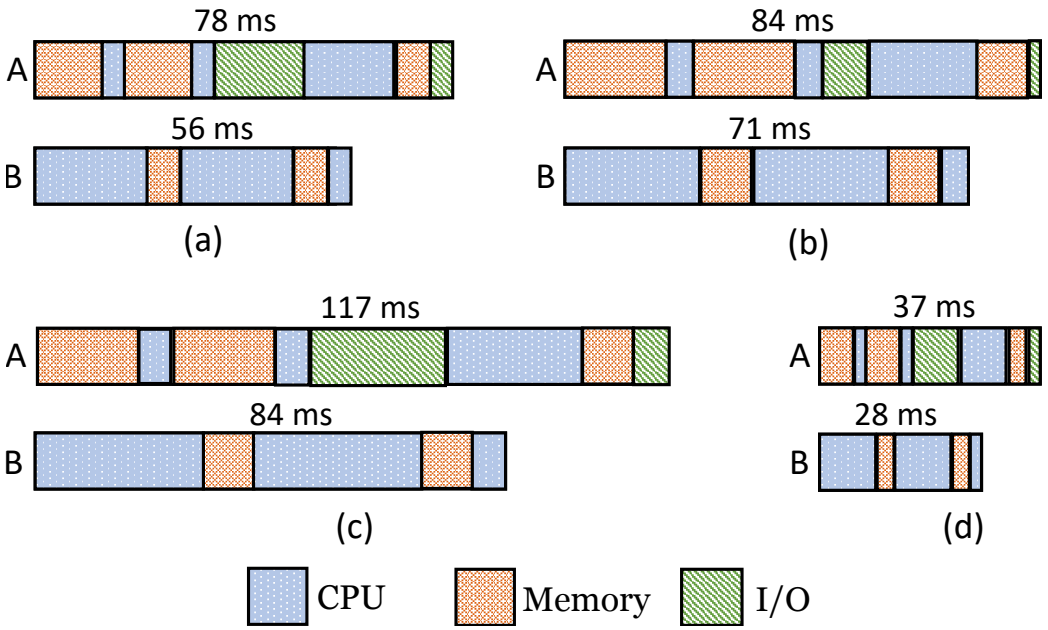


Figure 3.2: Two code segments A and B (split into CPU, memory, and I/O slices) running on (a) target device; (b) host device with different hardware component speeds; (c) host device with all hardware components executing 50% slower than the target device; d) host device with all the hardware components executing 2× faster.

In figure 3.2(c), the CPU and I/O on the host are adjusted to match the speed of memory, i.e., the processor runs 25% ($\alpha_{mem}/\alpha_{cpu}$) slower and I/O systems $3 \times (\alpha_{mem}/\alpha_{io})$ slower. This adjustment preserves the relative execution time of the two segments on the target device since all the slices scale by $\alpha = \alpha_{mem} = 1.5$. Figure 3.2(d) demonstrates another adjustment to the host hardware to conserve the relative execution time of code segments but this time, both segments scale by $\alpha = \alpha_{io} = 0.5$. For this case, CPU and memory are adjusted to operate 3× (α_{cpu}/α_{io}) and 2.4× (α_{mem}/α_{io}) faster, respectively.

Based on this idea, we design and implement a prototype of VCoz that configures the hardware components of the host system to simulate the causal profiling of target devices. Figure 3.3 presents the design of VCoz. First, VCoz estimates the performance scaling factor of each component on the host. To do so, it either compares spec of two hardware components or runs *performance tests* where micro-benchmarks stress distinct hardware resources on both systems and compares the results. For any pair of target and host, the stress tests are performed only once, and the measured scaling factors are stored in a *database* to eliminate the need for future access to the same devices.

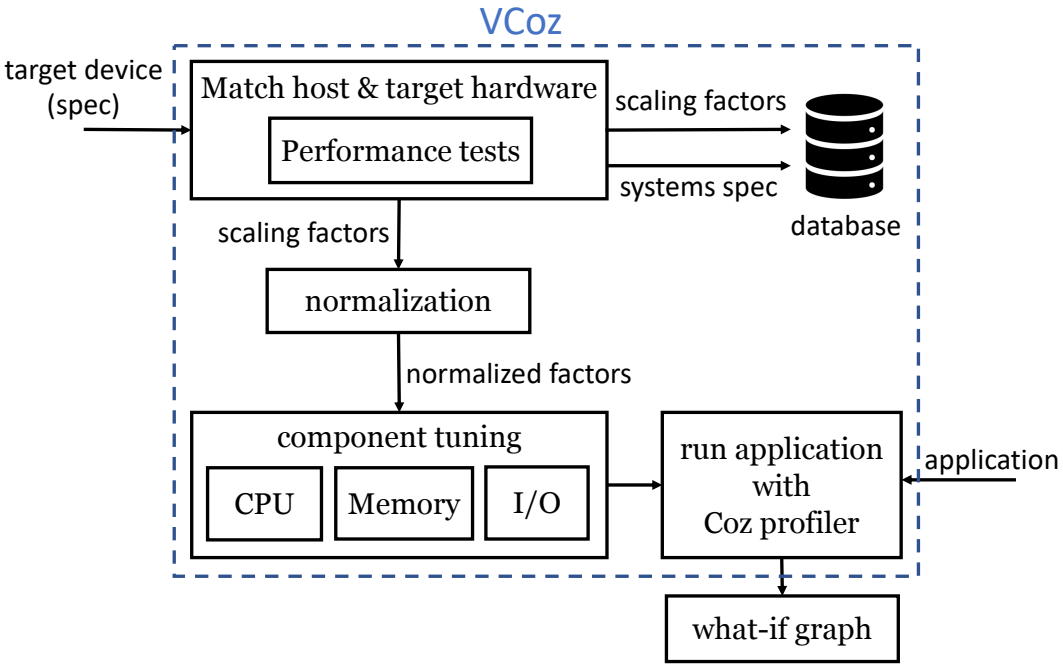


Figure 3.3: Design overview of VCoz containing inputs, modules, and outputs of each module.

After estimating the scaling factor of each hardware component, VCoz optionally normalizes them. *Normalization* handles two scenarios that arise on real devices. (i) In practice, sometimes the host system does not provide a knob for tuning a hardware component (e.g. memory). In this case, all the other resources are normalized by a scaling factor determined by the untunable component. (ii) The scaling factor of a component can be beyond the range of configurable values. For example, consider the CPU, and let's say it has to be scaled by

5×. However, the attainable CPU frequency on the host may limit the scaling to at most 4×. In this case, VCoz divides all the resources' (e.g. CPU, memory, and I/O components) scaling factors by 5.

Component tuning adjusts the hardware component according to the normalized factors. In our prototype, we normalize by memory since most of the systems do not expose a knob to change RAM operating frequency. Otherwise, VCoz changes RAM frequency either in BIOS or OS-level. For CPU, we modify the clock speed of all the cores using the `CPUfreq` governor in the Linux kernel [21]. For I/O, we adjust the interface of each I/O peripheral. Currently, VCoz configures the network interface using Linux traffic control utility (`tc`) [30] to restrict the uplink and downlink bandwidth of the system. It can also be easily extended to support other I/O components such as disk (using `hdparm` [29] in the Linux kernel). Once all the hardware components are tuned, VCoz runs the original Coz profiler on the application to generates a *what-if graph*. According to the theory, the host generated what-if graph will be the same as the one generated by Coz on the target device.

3.3 Porting Coz to Mobile Devices

The existing implementation of Coz is designed for and tested on desktop applications (e.g. applications running on x86 systems) that host the Linux OS. Therefore, the current version is not compatible with the majority of smartphones and mobile devices in the market. To improve the usability of this profiler among mobile users and also to use it later in this chapter for verification and accuracy measurements, we leverage Coz to support profiling of the applications that run on Android devices with ARM architectures. However, this extension is non-trivial and requires substantial re-engineering and troubleshooting of Coz and its dependent libraries. Figure 3.5 shows an overview of the Coz profiler and its interaction with the dependent libraries. The Coz bootstrapper in the figure interposes the entry

point of the target application (`__libc_start_main`) to initiate a profiler instance under the same process. This profiler instance uses third-party libraries (e.g. `libdwarf.so`) to read and process debugging information of the target application and C++ Pthread library to interpose Pthread APIs. It also invokes Linux `perf_event` system calls under the hood to monitor and profile the target application.

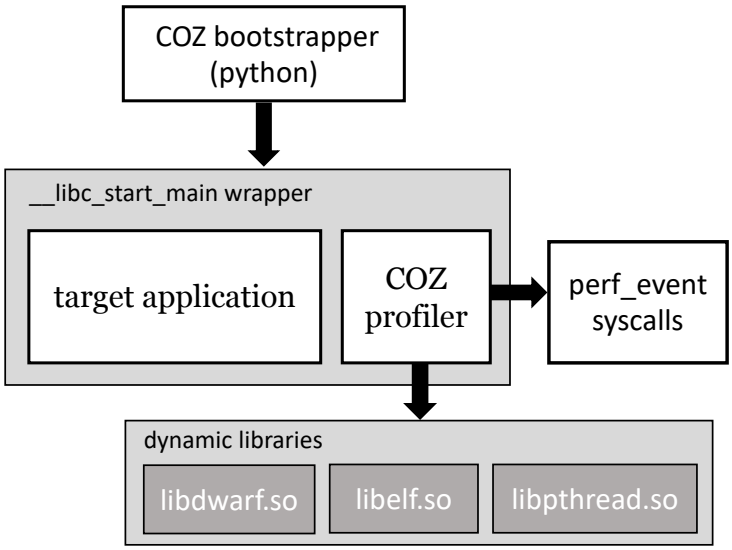


Figure 3.4: Overview of COZ profiler and dependent libraries.

To safely port Coz to Android devices, we rewrite its bootstrapper for two main reasons. First, the current code is not compatible with native Android apps. For instance, Google Android Bionic [25], the standard C library for the Android operating system, uses different entry points (e.g. `__libc_init`) and startup procedure than Linux standard C library. Second, most of the bootstrapper is written in Python which is not a standard language for Android systems. So, we rewrite the code in C++ and generate a native executable that can be safely launched directly by Android applications or via ADB shell without requiring a Scripting Layer for Android (SL4A).

Google's Android development kit (NDK) [14] is the official toolset for porting C/C++ programs to Android devices, however, not all of the C++ APIs and libraries for GNU/Linux are implemented by Android NDK. Therefore, the majority of our effort is to provide a

workaround for missing functionalities in the Coz code and third-party libraries. In some cases, we implement the missing functions (for example, `std::to_string()`, an STL function used in `libelf.so` and `libdwarf.so`) or replace them with compatible implementations. For instance, we use stack tracing APIs implemented in `stdio.h` instead of `execinfo.h`. The most challenging part, however, is to find a workaround for Pthread APIs since Coz interposes several Pthread APIs to handle thread suspension in the target application. Two fundamental issues with Pthread APIs prevent the current implementation of Coz to be ported to Android devices. 1) The entire Pthread API is not supported on Android devices. That being said, we remove all associated modules that belong to unsupported Pthread APIs (such as `sigwaitinfo, sigtimedwait, pthread_tryjoin_np, pthread_sigqueue, pthread_timedjoin_np, pthread_barrier_wait`) from the code. This pruning does not invalidate the functionality of the profiler since valid Android applications do not have calls to the unsupported Pthread APIs. 2) Pthread APIs are directly implemented through C/C++ library on Android rather than a separate Pthread library as in GNU/Linux systems. Thus, the existing implementation for loading and interposing symbols of the Pthread dynamic library, i.e. using `dlopen()` and `dlsym()` to load `libpthread.so`, is not compatible with Android. We fix this issue by providing Android-compatible wrapper functions for loading Pthread symbols.

Android NDK contains the implementation of the majority of the Linux perf APIs for Android devices. Since Perf APIs use kernel syscalls and read hardware counters, we have to validate and inspect their functionality on Android devices. Accordingly, we sample hardware counters used by Perf APIs in Coz on the Nexus 6P mobile with ARM v8 processor for multiple benchmarks in the Phoenix benchmark suite [131]. We then compare our log file with the log file from the same benchmarks on the x86 system hosting Linux OS and confirm the compatibility of the Perf APIs used in Coz as well as the functionality of the entire Coz sampling modules. Finally, we modify the Coz build and configuration files and pass specific switches and configurations for building ARM targets to the Android NDK compilers.

3.4 Experimental Setup

To validate our discussed hypothesis (Section 3.2) we run Coz [61] directly on the target device and compare it against VCoz for different applications. Figure 3.5 shows our testbed for this validation. The code snippet spawns two threads where each thread invokes a different benchmark. The top diagram in Figure 3.5 shows the program execution timeline for this testbed. We select the line corresponding to the slowest benchmark (i.e. line 10 which belongs to thread 2) for speedup. Consequently, Coz generates a what-if graph similar to the one shown at the bottom of the figure.

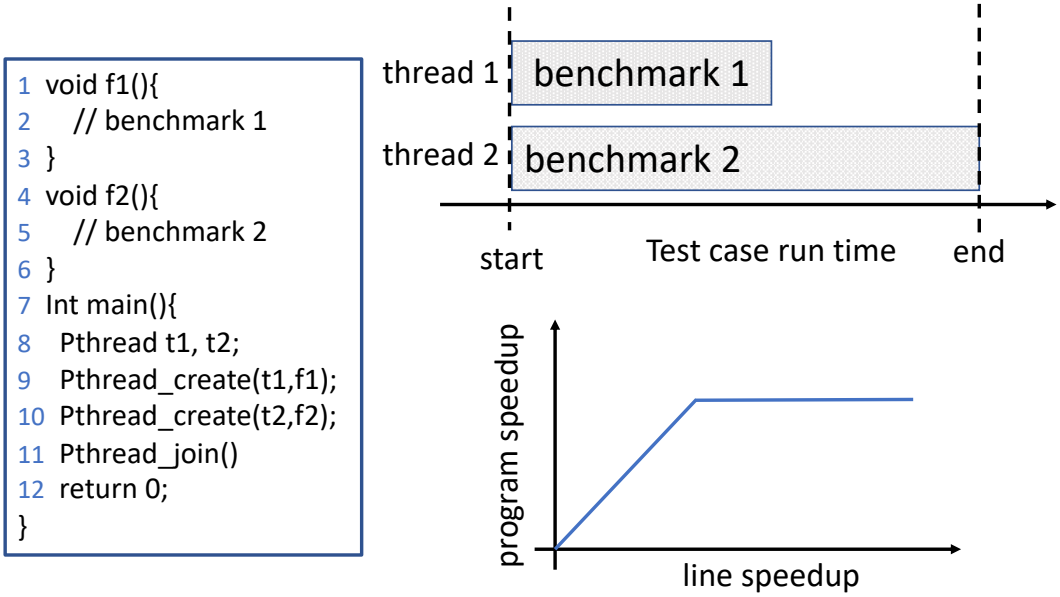


Figure 3.5: Description of test-case: the baseline code, program execution timeline (top diagram), and expected what-if graph generated by Coz profiler (bottom diagram).

We integrate benchmarks with different types of workloads in our testbed: CPU-intensive (LU and Cholesky decomposition from Splash [133]), memory-intensive (stream benchmark [36]), and I/O-intensive (network client-server benchmark). Note that benchmarks can have a mix of all three types of instructions (CPU, memory, and I/O). For example, Cholesky decomposition contains memory load/store operations to read/write matrices. However, to examine the impact of hardware component tuning in VCoz, we consider benchmarks to

primarily stress one component.

Benchmark	**Nexus 6P**	**MacBook Air**	**ratio (α_{cpu})**
Matrix Multiply	3.1 s	1.5 s	2.1
FFT	56 ms	23 ms	2.4
LU	2.3 s	1.0 s	2.3
Word Count	38 s	16 s	2.3
Cholesky	1.1 s	450 ms	2.4
PCA	690 ms	300 ms	2.3

Table 3.1: Execution time of 6 CPU-intensive benchmarks on Nexus 6P and MacBook Air.

The target device is Nexus 6P with quad-core ARM Cortex-A53 + quad-core ARM Cortex-A57 processor, and the host device is MacBook Air 2.2 GHz Intel Core i7 processor. Both systems have Low Power Double Data Rate (LPDDR) dual channel memory operating on 1600MHz, so they are expected to have similar memory performance, i.e., $\alpha_{mem} = 1$. The processors, however, have different architecture, so VCoz needs to find the CPU scaling factor. For this reason, VCoz runs multiple CPU-intensive benchmarks (from Phenoix [131] and Splash [133] test suites) and compares the execution time on the host and target devices as shown in Table 3.1. We observe that the host x86 processor computes approximately 2.3× faster than the mobile ARM processor (i.e., $\alpha_{cpu} = 2.3$). We consider the network interface as an example of I/O in this work. For each experiment, we try 20 different speedups from 0 to 100% and 4 to 8 profiling runs for each speedup.

3.5 Results

We compare the results of virtual causal profiling (VCoz) with the causal profiler (Coz) that runs directly on the device to assess the functionality of VCoz and evaluate the accuracy of our prototype. Figure 3.6 shows the corresponding what-if graphs in purple and blue lines, respectively, for various combinations of benchmarks that stress different hardware resources. Additionally, we also compare them against the results generated by Coz on the host x86

system (yellow line) to evaluate the impact of hardware tuning.

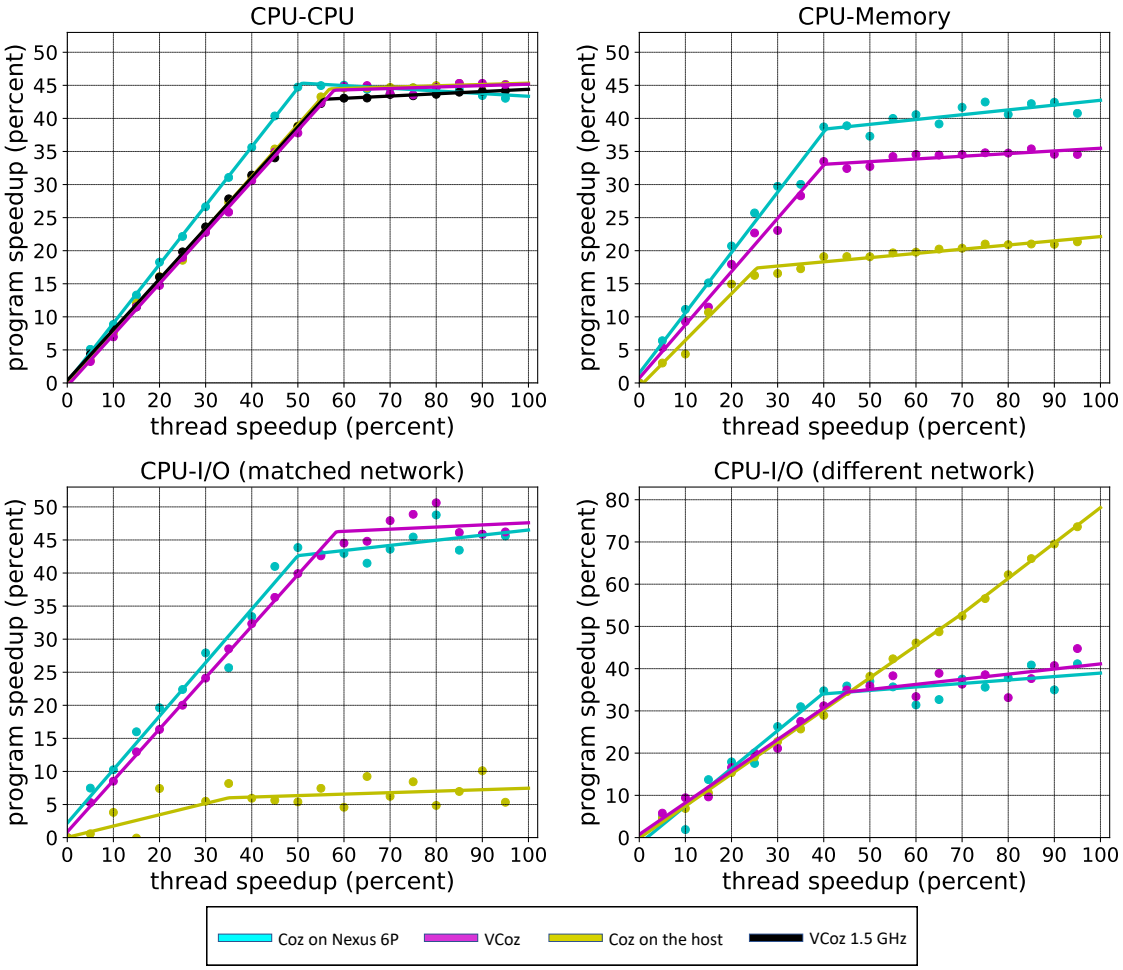


Figure 3.6: Comparison of VCoz and Coz on host and target devices for different test cases. Plots from left to right: 1) CPU frequency tuning on a program with two streams of compute-heavy code. 2) Memory and CPU heavy code segments. 3) I/O and CPU heavy code segments. Both devices are connected to 100 Mbps network connection. 4) I/O and CPU heavy code segments. Host is connected to 100 Mbps network connection and Nexus 6P is connected to 17.7 Mbps.

3.5.1 CPU and Memory Test-cases

CPU-CPU. We first consider benchmarks with a similar workload in the testbed (section 4.5). We run two different CPU-intensive benchmarks namely, Cholesky and LU decomposition, concurrently and the results are shown in the leftmost plot in Figure 3.6. As we can

observe, VCoz predicts the result of causal profiling with at most 13.6% variance. Moreover, the graphs generated by Coz on the host (no CPU frequency tuning) and VCoz with the host CPU frequency tuned to 1.5 GHz are comparable to VCoz with accurate CPU tuning (900 MHz), which indicates that CPU tuning is not effective for this test case. This supports our theorem since scaling CPU by any factor would homogeneously scale both execution paths, maintaining the relative speed of code segments and hence producing identical what-if graphs.

CPU-Memory. In this test case, we run the stream benchmark on one thread and LU decomposition on the other to evaluate a combination of two different workloads: memory-intensive and CPU-intensive. According to Figure 3.6, VCoz predicts the program speedup on the target device with less than 15.8% error on the host system while Coz incurs more than 50% error. Note that CPU and memory have different scaling factors in our experiments. Therefore, this test case highlights the limitation of Coz in profiling devices where CPU code segments execute at different relative speeds compared to memory code segments, thereby violating the necessary condition for causal profiling.

3.5.2 I/O Test-cases

Now we evaluate the behavior of VCoz when the program has I/O operations. Here we run a network-heavy benchmark (server-client data streaming) as an example of an I/O-intensive workload. On the concurrent thread, we run a CPU-intensive benchmark (LU decomposition).

Matched network. First, both systems are connected to the same network (100 Mbps), which allows VCoz to match the I/O speed without adjusting the network module of the host system (i.e., $\alpha_{io} = 1$). So, it only adjusts the CPU frequency. As we can infer from the third plot in Figure 3.6, VCoz predicts the result of Coz on the device with high precision

(less than 9.1% variance). Meanwhile, Coz is unable to uncover the potential impact of optimizations on the target device since it incorrectly predicts marginal program speedup.

Different network. Now, we connect the Nexus 6P to a slower network connection, 17.7 Mbps. This is the global average network connection for mobile devices in 2020 [5, 16]. Since the I/O speed of the target device is modified, VCoz reapplies component matching. The I/O speed is now 5.6× slower, and the CPU computing speed is 2.3× slower. If VCoz normalizes by the I/O scaling factor, the CPU has to run 2.4× faster. Given the operating frequency of the host system is (2.2 GHz) and the maximum available CPU frequency is 3.1 GHz (in turbo boost mode), VCoz cannot fulfill this scaling factor. However, if we normalize by the CPU scaling factor, the network speed has to reduce by 2.4×, and VCoz can satisfy this by adjusting the network interface of the host system to limit the bandwidth to 41 Mbps. Figure 3.6 shows how remarkably VCoz predicts the actual causal profile with less than 11.4% error. Meanwhile, Coz cannot detect the behavior of the critical path (no changes in the critical path) in the range of speedup because of missing characterization of the underlying hardware of the target device.

3.6 Related Work

Profilers are an important tool for performance analysis of applications. Tools such as gprof [101] and Oprofile [33] along with the Linux built-in hardware counter profiling tool, perf [46], and the equivalent for Android, simplePerf [35] are among the popular profiling tools for desktop and mobile developers. These tools rank code by its contribution to total execution time, however, code that runs for a long time is not necessarily a good candidate for optimization.

Besides profilers, developers exploit simulators for performance analysis for cross-platform

application development and optimization. Cycle-accurate instruction set simulators (ISS) such as gem5 [70] and QEMU [69] and full-system emulators such as Android studio emulator [13] and Appetize [15] for iOS generate relatively accurate timing and power consumption for system-level performance analysis. However, simulation on these platforms is considerably slower which makes it infeasible for comprehensive what-if analysis [143, 138, 90] which typically requires a large number of experiments to explore the hardware and software design space. Multiple prior efforts have attempted to enhance the functionality and speed of ISS for performance and power analysis of mobile platforms [77, 100]. Nonetheless, they are still slow making them infeasible for causal profiling [143].

3.7 Conclusions

We present a theoretical formulation for causal profiling and extract a necessary condition for virtual causal profiling. According to the theory, the result of causal profiling only depends on the relative execution speed of the different code segments. Therefore, we design VCoz that enables virtual causal profiling by emulating the hardware configuration of say, mobile devices on a laptop/desktop for causal profiling by controlling the relative performance of various resources, such as CPU, memory, and network. We implement a prototype of VCoz and evaluate it on multiple benchmarks with a combination of different workloads (CPU-intensive, memory-intensive, and I/O-intensive). Our results show that VCoz can predict the results of causal profiling with significant accuracy by tuning different hardware components. For example, VCoz predicts the outcome of the Coz profiler on an Arm-based Nexus 6P mobile device with less than 16% variance on an x86 laptop while the original Coz profiler misses predicting optimization opportunities or generates inaccurate what-if graphs. As a result, VCoz advances the state-of-the-art in designing practical profilers that are much needed for scalable cross-platform application development.

Chapter 4

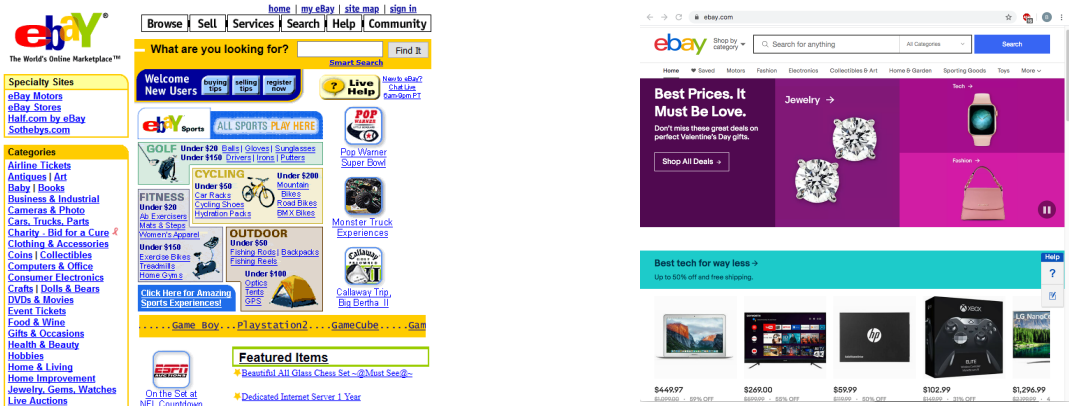
Performance Characterization of Third-party Ads

Online advertising is another interesting subject that has raised many performance questions in recent years. As an example, it is not clear how much ads contribute to the performance of websites. The key to answering these questions is to apply performance characterization (or workload demystification) which is a domain of PA. The diversity of today's display ads and their indirect methods of delivery and integration on the website makes this characterization challenging. The existing workaround for workload characterization of web ads is primarily based on *block-and-measure*, i.e., using ad-blockers. However, the overhead of ad-blockers and issues such as site-breakage and ad-blocking circumventions guarded by websites limit the scope of using ad-blockers for workload characterization. In this chapter, we again use profiling fundamentals to build a tool that accurately demystifies the performance of web ads. Our proposed block-free solution leverages profilers to *record* the state of the program (e.g., website loading) at runtime and *replay* it for offline measurements and characterization.

4.1 Introduction

Online advertising (essentially display ads on websites) has proliferated in the last decade to the extent where it is now an integral part of the web ecosystem with a multi-billion dollar market [82, 85, 24]. Today, publishers display multiple advertisements (or ads) through pop-ups, banners, click-throughs, iframes, widgets, etc., to monetize their websites and web apps. The majority of these ads neither come from the publisher (website) nor a specific domain. They are delivered through a chain of third-party content providers (such as ad providers, syndication agencies, ad exchange traders, trackers, and analytics service providers) who are part of a complex ad network on the server-side [95]. The current ad delivery method forces publishers to embed unknown third-party content (such as JavaScript or HTML) on their website which could jeopardize user privacy and security. There have been several studies in recent years to locate the untrusted sources and malicious ad contents [109, 68, 95, 106, 84, 65]. Accordingly, different blocking and evasion policies have been devised to guard against such malware and aggressive tracking [68, 97, 162, 157]. While user privacy and security are of paramount importance, it is not the solitary concern of the worldwide web community. Online advertising also has a direct impact on website performance (eg., page load time) and in turn user satisfaction.

Web ads have become more diverse and complex keeping up with the pace of advances in web design. Figure 4.1 compares advertising on `ebay.com` in 2002 and 2020. As we can observe, in the past, ads only included hypertext and images. However, today's online ads comprise of JavaScript, iframe, animation, multimedia, etc. Evaluating and displaying these dynamic ad contents demand increased computation from the browser and competition for the user's device resources. Coupling this observation with recent studies [123, 146, 111] that show that most of the page load time is spent on computation activities in modern browsers raise three key questions:



(a) `ebay.com` in 2002 (b) `ebay.com` in 2020

Figure 4.1: Evolution of ads on the web. (a) Early web ads contain text, image, and hyperlink. (b) Today's complex and dynamic web ads (rotating on top of the website) contain JavaScript, animation, multimedia, and iframe.

- *How much do ads increase the browser's page loading workload?*
- *What type of web documents and browser activities contribute most to this workload?*
- *What kind of sources deliver web-ads and how much do they contribute to its performance cost?*

Gaining insight into the above questions and understanding how much ads contribute to the breakdown of different activities in modern browsers can inform the design of efficient ads and optimizations targeting those specific activities. Unfortunately, only a handful of studies [128, 94, 92, 73] have been devoted to the performance analysis of ads, yet many such important open questions remain to be answered.

Previous studies revolving around the performance analysis of ads lack a comprehensive examination for at least the following reasons. First, the majority of them concentrate on the network data traffic overhead, neglecting the performance cost of browser computation activities such as rendering activities [92, 128]. Second, prior efforts fundamentally share the same approach for quantifying the performance of ads. They use ad-blockers to block websites' ad contents and assess the performance overhead via comparison with *vanilla* run

(no ad-blocking). This approach, however, is prone to inaccuracy as it does not take into account the intrinsic overhead of the ad-blocker. Our measurement of over 350 websites shows that Adblock Plus [9]–the most popular ad-blocker–adds **32% overhead** (median) to the page loading even though it reduces the overall page load time by aggressive content blocking. Furthermore, ad-blockers lead to site breakage and undesired app functionality, particularly, with the prevalence of anti-ad-blockers [94, 96]. Finally, the ad-blocker approach suffers from an inability to perform comprehensive and fined-grained performance analysis. This stems from the way ad-blockers operate where ad-related content is blocked as early as the initiation of network requests. Thus, subsequent ad-related activities such as content parsing, descendent resource loading, and rendering remain invisible for inspection.

In this chapter, we investigate the performance overhead of all types of ad-related content by crawling over 500 websites on different systems (laptop and smartphone). Unlike previous efforts, we take a novel approach based on in-browser profiling that does not rely on ad-blockers. Our methodology allows the browser to automatically fetch and evaluate ads' performance at scale. It correlates the browser's computation and network activities to the associated ad contents and quantifies the added cost of ads. We develop a tool, *adPerf* based on our technique for the Chrome browser. The key challenge we encountered is how to align the performance cost with individual components within an ad (e.g., image and JavaScript code), and we address this through a carefully designed resource mapper (section 4.3).

We break down the performance overhead to individual requests and content types through a resource mapping technique. This procedure contrives a more robust and detailed performance analysis. To assess the impact of the platform on the performance overhead of ads, we compare our performance evaluations on a laptop connected to high-speed WiFi with a smartphone connected to a cellular network (Section 4.6). In Section 4.6.3, we demystify and track down ad components on the publisher and characterize the performance overhead considering the origin of ads and how they are delivered to the publisher. To the best of our

knowledge, this is the first time such an experiment has been conducted.

adPerf and the detailed measurement results are available at `https://gitlab.com/adPerf/adPerf`. adPerf can be leveraged by web researchers and developers for deeper performance evaluation of ads and reducing the overhead of ads. Specifically, we present two compelling use-cases of adPerf for the efficient design of ad intervention and to improve the performance of ads in Section 4.6.5. Additionally, we compare our takeaways with previous studies in the literature (Section 4.7), and discuss similarities and inconsistencies.

4.2 Ad Blocking and Performance Analysis

Ad blocking is a defense mechanism against advertising and tracking that is wildly deployed by end-users. According to Statistica [32], the global number of clients with connected devices to ad blockers is steadily increasing, and more than a quarter of Internet users in the US were blocking ads in 2019 [7]. Popular ad blockers such as AdBlock [8], Adblock Plus [9], uBlock [37], and Ghostery [23] install as browser add-ons and use filter lists to block web ads and trackers. While user privacy and security are crucial, even ads that are safe and not tracking users can have a significant performance impact that has cascading effects on user satisfaction and Internet costs. Some notable studies [73, 128, 92, 151, 134] lean on ad blockers to measure the performance cost of web ads. The key distinction between our approach and prior efforts is that *we do not rely on ad blockers and content-blocking for performance analysis of ads* for three main reasons:

Overhead. Multiple studies [136, 92, 94] report ad blockers themselves have significant performance overhead due to exhaustive filter-list matching, tracking services of their own, and running background scripts. Our results also affirm this observation. We analyze AdBlock Plus by creating a modified version that still performs all of the content filtering operations

without actually blocking any of the content. We calculate the overhead imposed by these filtering operations by measuring the difference in page load times from the modified version of Adblock Plus to the vanilla instance of Chrome. Figure 4.2 shows the overhead of Adblock Plus on 350 webpages in our corpus (see section 4.5). According to the figure, for half of the websites, Adblock Plus adds more than 32% overhead to the page loading due to excessive and CPU-intensive filter rule matching and add-on background activities. Although it may ultimately reduce page loading workload and network cost by aggressive content blocking, it's an inaccurate tool for studying the performance impact of ads.

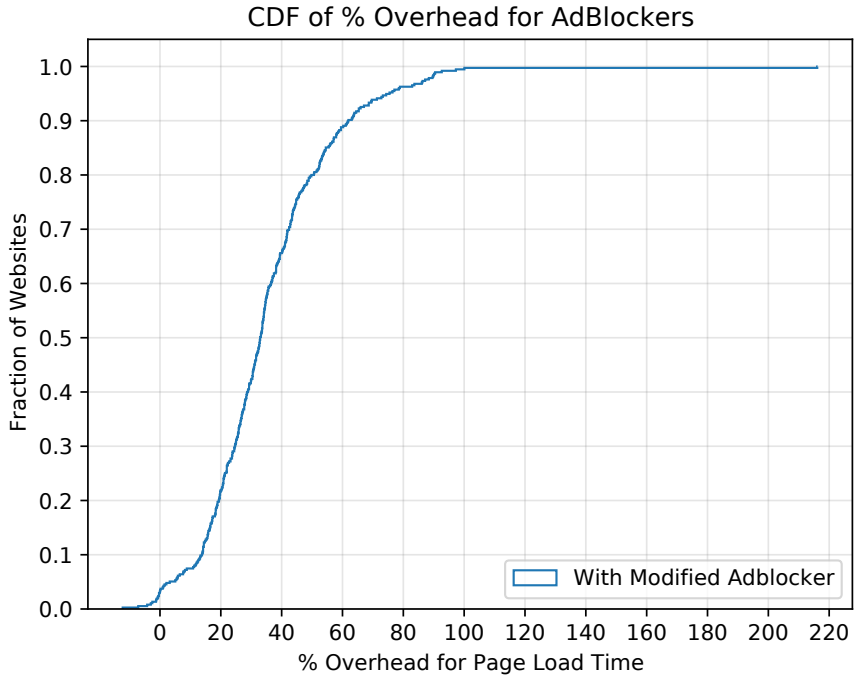


Figure 4.2: CDF distribution of AdblockPlus overhead on the page loading of 350 webpages.

Functionality. As ad blockers become a threat to the "free" web business model, many websites prevent displaying their content to the visitors that use ad blockers. In this case, the publisher includes a script such as IAB ad block detection script [6] that monitors the visibility of ads to DEAL (Detect, Explain, Ask, Limit) with ad blockers [115]. Typically, when the publisher detects a hidden or removed ad, it immediately stops loading the website by displaying a popup that asks the visitor to turn off the ad-blocker. Figure 4.3 shows a

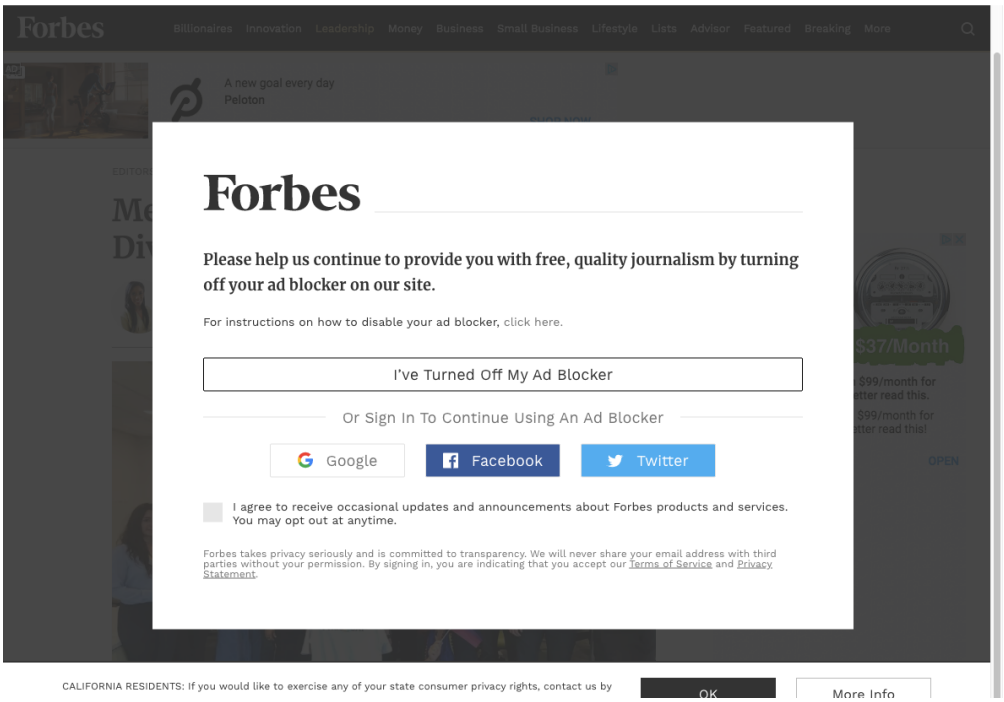
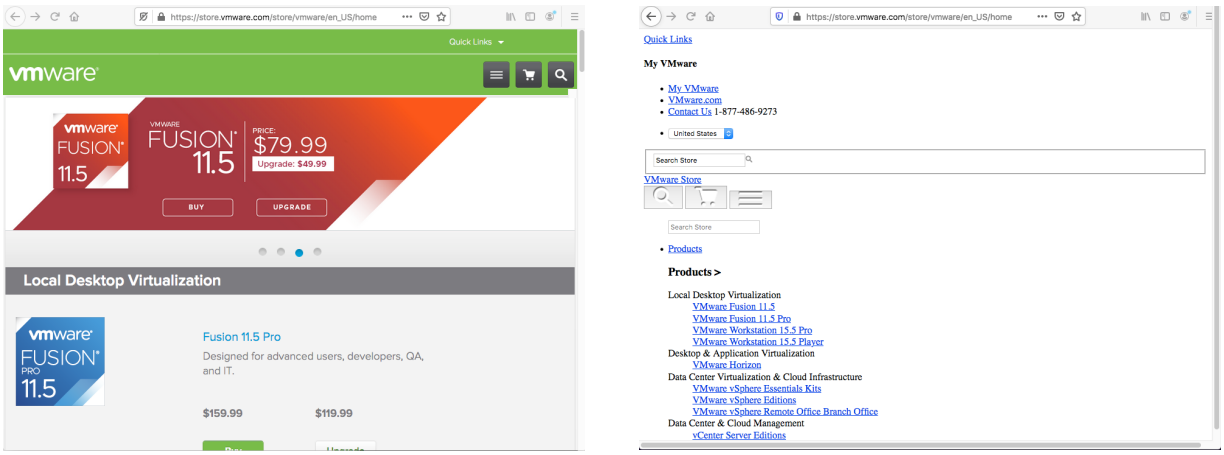


Figure 4.3: Snapshot of `www.forbes.com`. This website prevents loading contents if visitors attempt to block ads.

snapshot of the content-blocking of `www.forbes.com` when ad blocker is on. As reported, a large portion of the web, 6.7% of Alexa top 5000 [118] and 16.3% of the top 1000 popular live streaming sites [130] use this anti-adblocking system.

Besides, content-blocking can also lead to site breakage and other undesired app functionality [96]. This breakage can range from a dysfunctionality in part of the website (e.g., not displaying login popup) to the breakdown of the entire website layout. For instance, figure 4.4 shows a snapshot of `www.store.vmware.com` when Mozilla's ad and tracking protection is turned on. Furthermore, a large number of websites employ ad blocking circumvention to evade from ad blocking. For instance, `www.thoughtcatalog.com` and `www.cnet.com` obfuscate advertising URLs when they detect that the ad blocker is on. As a result, the resources are translated to the local servers and eventually displayed on the page. In all of the above cases, performance analysis of ads through ad blocking is infeasible which limits its scope.



(a) Before content blocking

(b) After content blocking

Figure 4.4: Snapshots of `www.store.vmware.com`. The layout of the page is broken due to content blocking.

Fine-grained analysis. Ad blockers block content as early as the initiation of network requests, which results in two drawbacks. First, it prevents fine-grained performance analysis at the browser level because activities such as content parsing and rendering related to the blocked ad become invisible for analysis. Hence, the current body of work focuses on the network data traffic overhead, neglecting the in-browser computation overhead of ads. Second, because the content is blocked at the network request, resources that are further requested by the blocked document during page loading become invisible for inspection. For example, when an ad exchange[1] (e.g., Google AdSense) script is blocked, the source(s) of the blocked ads is hidden.

Our approach addresses the above limitations and enables an in-depth performance analysis of ads without adding significant overhead. As a result, our measured cost is more reliable and reveals some anomalies with previous studies that we discuss in section 4.7. Plus, our framework measures performance cost of ads on every website. This is highly substantial in term of functionality because we examine only 15% of our test corpus ($\approx$ 50 websites) and discover 10 websites to have one of the discussed issues with ad-blocker. Ultimately,

[1] A platform for buying and selling of advertising inventory from multiple ad networks through real-time bidding.

we do not block any content, this gives us the ability to correlate the performance cost of ads to the sources (domain analysis) as well as break down the computation cost at a finer level of granularity (browser activities) which have not been studied before. We present our first-of-a-kind findings from this fine-grained characterization in Section 4.6.

4.3 Methodology and adPerf

To distinguish the performance cost of web ads from the primary content (non-ads), we apply a systematic approach. First, we *extract* all browser activities that are associated with the page loading process. Second, we *identify* which resource (i.e., a web document) explicitly or implicitly initiates each browser activity. Third, we *classify* activities into ads and primary content based on the resource type initiating the activity. Finally, we *measure* the total execution time spent on each class of activity as a performance index distinguishing the workload in each class.

To realize the above methodology, we design and implement a tool, *adPerf*, for the Chrome browser. Note that adPerf can be extended to support other browsers since the same technique applies to all browser architectures. Figure 4.5 shows the design of adPerf. Below, we describe the main modules of adPerf – crawler, parser, resource mapper, and graph builder.

4.3.1 Crawler

The first module in adPerf (top of the figure) is a *crawler* (`Node.js` script) that sets up the headless Chrome and crawls websites. The crawler uses the Chrome remote protocol APIs [17] under the hood to interact with the browser and streams Chrome traces [51] to a file. Chrome traces are primarily used for profiling and debugging the Chrome browser and are low-overhead. Tracing macros cost a few thousand clocks at most [51], and the logging to

file happens after the page is loaded. Chrome traces are capable of recording intermediate browser activities, including page loading activities in the Blink rendering engine and V8 JavaScript engine with microsecond precision. Each trace contains information about the associated activity, such as thread id, activity name, function arguments, etc. Below is an example trace for a *Scripting* activity:


```
{"pid":54,
 "tid":35,
 "ts":81407054,
 "ph":"X",
 "tts":119412,
 "dur":839,
 "cat":"devtools.timeline",
 "name":"EvaluateScript",
 "args":{"data":{
        "url":"https://www.google-analytics.com/linkid.js",
        "lineNumber":1,
        "columnNumber":1,
        "frame":"EFF8B95C2"}}}
```


Additionally, the crawler intercepts network requests, i.e., `onBeforeRequest` event, and extracts the header and body of every HTTP request. This data is necessary for resource matching.

4.3.2 Parser

When the website is loaded, the raw Chrome traces are fed to the *parser* as shown in the figure. The adPerf parser does two tasks – pruning and data extraction.

Pruning. The parser goes through the traces and extracts all page loading activities and prunes the browser-dependent ones (such as browser garbage collection and inter-process communication activities). We use the same subset of traces that robust tools such as

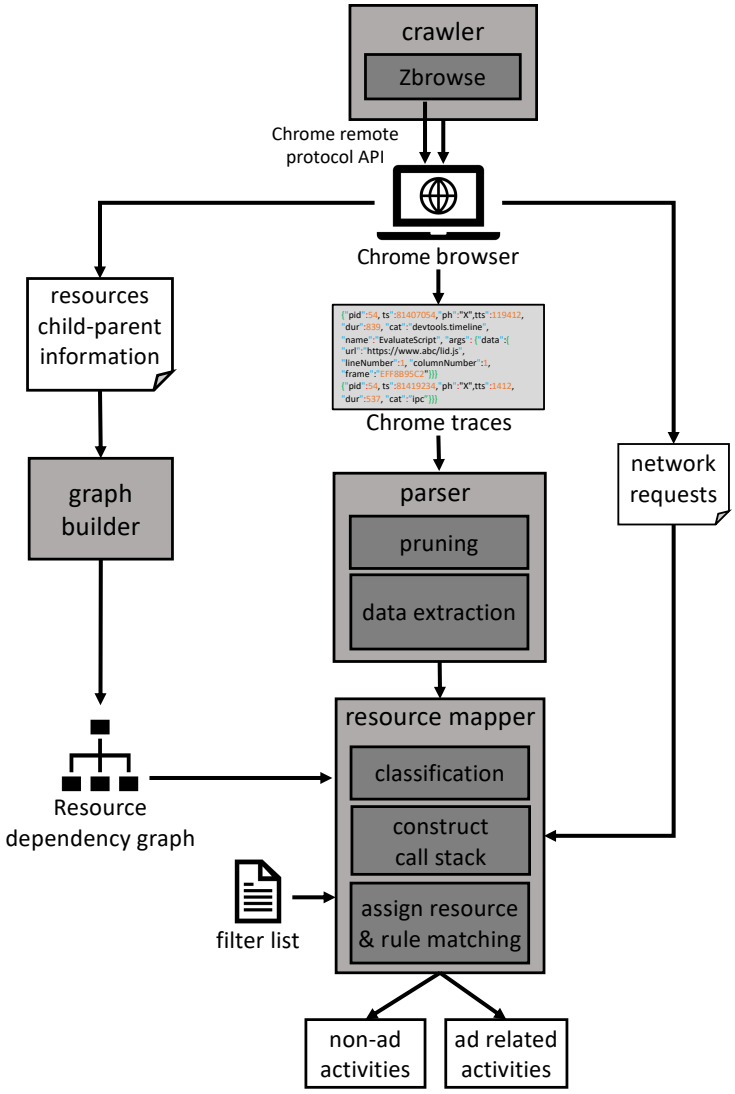


Figure 4.5: Design of adPerf. The four core modules are crawler, parser, resource mapper, and graph builder that are shown with dark boxes.

Chrome devtools timeline [18], Google Lighthouse [28], and COZ+ [20] collect for performance analysis and page loading workload characterization. The resulting activities are associated with one of the six browser stages shown in Figure 2.1. For instance, the parser considers every trace connected to script evaluation, V8 script compiling, V8 execution, callback functions triggered by browser events (or timeouts) among others as part of the Scripting stage.

Data extraction. For each activity, the parser extracts the following data: start time, end

time, relative stage, thread and process ids, and function arguments if they contain resource information. This data is necessary to construct the call stack and attribute activities to resources.

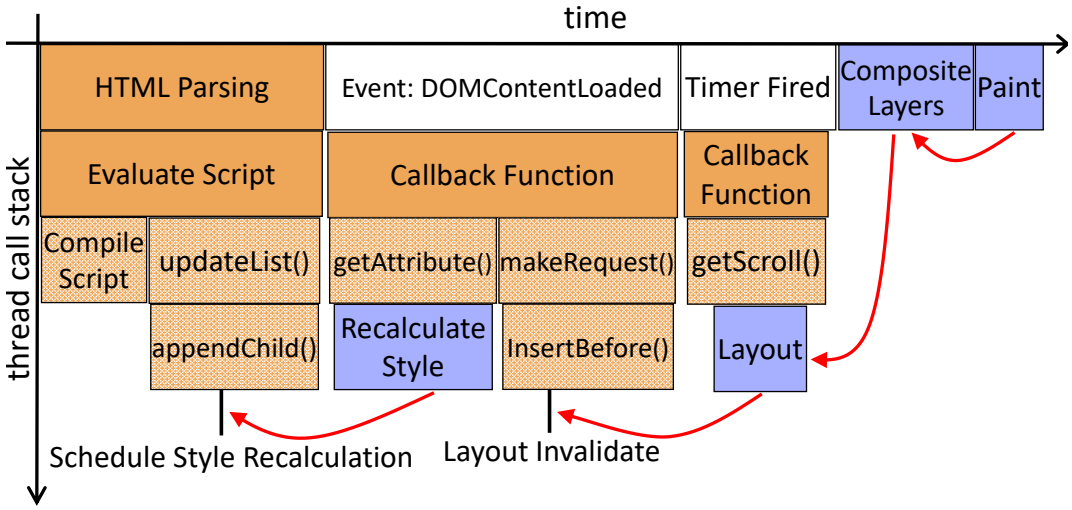


Figure 4.6: Call stack timeline for a Chrome thread constructed by adPerf resource mapper. The resource mapper assigns a resource to each activity using the information in the traces (orange activities with solid texture) and call stack (orange activities with dotted texture) for parsing and evaluation activities and tracks initiator for tree manipulation and rendering activities (purple activities).

4.3.3 Resource Mapper

Once the traces are parsed and categorized, this data and network information extracted by the crawler are input to the *resource mapper*. The task of the resource mapper is to assign each activity to an associated resource. Unfortunately, we observed that a significant number of traces (about 30%) do not contain any resource information. In such cases, the resource mapper has to derive this relation.

To address the above challenge, the resource mapper first builds a call stack of activities for every thread by tracking the start time and end times of activities executed by each thread. Figure 4.6 shows the call stack timeline for a sample activity for a browser thread

where activities are shown with boxes. After constructing call stacks, the resource mapper classifies activities into two groups – *parsing and evaluation* and *tree manipulation and rendering*. The former contains activities that *explicitly* relate to a resource such as HTML parsing, image decoding, stylesheet parsing, and JavaScript evaluation that directly operate on a document. Activities belonging to this group are colored orange in the figure. The latter contains activities that *implicitly* relate to a resource. These include activities in styling (except stylesheet parsing which belong to the former group), layout, composite, and paint stages that deal with the browser's intermediate data structures (trees) and display. Purple activities in the example belong to this group. Finally, the resource mapper finds the corresponding resource for each activity group as follows.

Parsing and evaluation. For the majority of the activities in this group, the resource mapper extracts the resource file information from function parameters extracted by the parser. Orange activities with solid texture such as *HTML Parsing* and *Callback Function* in Figure 4.6 are examples of activities where we can determine the document on which they parse or evaluate from frame id and resource information in their traces. However, a small number of activities in this group do not contain any resource information. For activities with unresolved resource files (activities shown with an orange color and dotted texture in the figure), the resource mapper uses the constructed call stack and follows their ancestors and associates them with the caller's resource file. For example, *appendChild* JavaScript function is called by *updateList* and this function along with *Compile Script* activity are invoked by *Evaluate Script* activity that is previously assigned to a JavaScript document.

Tree manipulation and rendering. For this group, we have to distinguish between the different resources that implicitly trigger the activities that belong to this group. For styling activities, we observe that Chrome recalculates styles after the *Schedule Style Recalculation* event is fired. As seen from Figure 4.6, this event is fired in the middle of *parsing and evaluation* of a resource (typically a JavaScript document) that attempts to modify the DOM node

style. We track the call stack for this event to the initiated *parsing and evaluation activity* and associate this styling activity to the triggered document. Similarly, for layout, Chrome updates the layout tree when the *Layout Invalidate* event is fired. In our example, this is fired when the command *this._util.elem.innerHTML=e* is executed in the *InsertBefore()* function. We use a similar procedure as styling to associate layout activities to the initiating resource from the call stack of the *Layout Invalidate* event.

Note that the browser does not always update the style and layout of nodes immediately after the events are triggered. Depending on the priority of other activities in the task scheduler queue, the browser might dispatch these activities later. As a result, when a resource triggers one of these two events (*Schedule Style Recalculation* or *Layout Invalidate*), a second resource may fire these events again before the browser updates the tree. In this case, we consider the first resource as the initiator since the tree will be traversed and updated even in the absence of the second resource. Chrome tends to composite and/or paint immediately after styling or layout which leads to repaint. Therefore, the associated resource for the composite and paint activities simply derives by following the chain to the last executed styling or layout activity as shown by the red arrows in the figure.

Once page loading activities are mapped to the corresponding resources, adPerf uses network data from the crawler to link the resources to the associated network requests (i.e., URLs). Then it uses a filter list to distinguish between ad resources and non-ad resources. We use EasyList [22], the primary and most popular filter rules list for advertisements for our experiments. However, users can also provide their own custom filter rules. adPerf employs adblockparser (an optimized python package [10]) to match the URLs against filter rules. One might think that since our methodology uses an identical rule matching procedure to ad blockers, it might incur a similar overhead. However, this is not the case since rule matching in adPerf is passive and does not steal computation cycles from the page loading process. Finally, adPerf reports the execution time of the page loading activities categorized by ads

and non-ads.

4.3.4 Graph Builder

There exist dependencies between resources on the website. For instance, let's say a website downloads a JavaScript file from a third-party domain. In this file, it can further request an image or an HTML document from another domain, and this chain can go deeper. To evaluate the performance cost of different sources such as ad domains and to further evaluate their trustworthiness requires first tracing this resource dependency chain and building a *resource dependency graph*.

We extract the dependency between resources of the websites using Zbrowse [40]. Zbrowse uses Chrome devTools protocol and allows us to instrument, inspect, and debug the Chrome browser. It also generates the child-parent relation for every network request. We embed Zbrowse in the adPerf crawler module as shown in Figure 4.5. This way, we can extract the resources child-parent data at the same time when we crawl the websites. The *graph builder* uses Zbrowse's output and constructs the dependency graph for resources. In cases where third-party JavaScript gets loaded into a first-party context and makes an AJAX request, the HTTP referrer appears to be the first-party. We follow [95] and allow the graph builder to conserve this relation and include the URL of the third-party from which the JavaScript was loaded. Since one resource can, in turn, request multiple resources, the constructed graph has the shape of a tree rather than simple chains of dependencies.

Figure 4.9 shows this graph for an example website, `www.cnn.com`. Here, we combine the resources from the same domain (at each level) into one node for easier visualization. The root node is the publisher and the remaining nodes are referred to as third-party domains. For differentiation, we color ad nodes (domains that deliver at least one ad resource) red and non-ad nodes (domains without any ad resources) blue in this graph. As we can see from

the figure, a considerable number of third-party domains are ad nodes. This is a concerning finding since typically publishers are not aware of the contents delivered by third-party websites. Generally, publishers trust the first-party domains (in the first-level of the tree) but those websites might deliver their contents from another website or chain of websites that are not verified by the publishers. We investigate the prevalence of such third-party ad domains, their performance cost, and trustworthiness in section 4.6.

4.4 Validation of adPerf

adPerf is a first-of-a-kind performance analyzing tool that measures the fine-grained performance overhead of web ads at the granularity of the browser's major stages. In the absence of tools with similar functionality to serve as a baseline, it is challenging to test and validate adPerf. Chrome DevTools [18], a set of web developer tools built directly into Google Chrome, provides sufficient and useful profiling data, including a breakdown of the browser workload into stages. However, the caveat is that the reported breakdown is for the entire page content, and it does not differentiate between ads and main content. Therefore, we devise the following experiment to exploit Chrome DevTools to calculate the performance of ads on a webpage and validate adPerf.

In our validation experiment, we first measure the total workload of a test page with Chrome DevTools. Then we instrument the test page by cloning every ad element on the page and re-measure the total workload. If the cloning is perfect, the added workload will present the performance overhead of web ads. This experiment validates two main objectives:

1. How precisely does adPerf measure the computation workload (irrespective of ads and main content) and classify them by the browser stages? This is achieved by comparing adPerf's reported *total* workload and its breakdown with Chrome DevTools data.

2. More important, how well does adPerf distinguish the main content workload from the advertising workload? This is validated by comparing the performance of the added workload (which represents only ads) measured by Chrome Devtools with adPerf's reported ads performance cost.

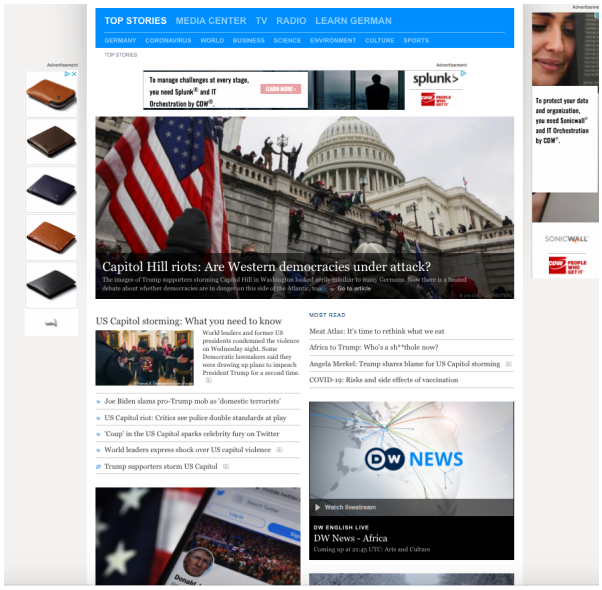


Figure 4.7: Snapshot of www.dw.com before cloning ads.

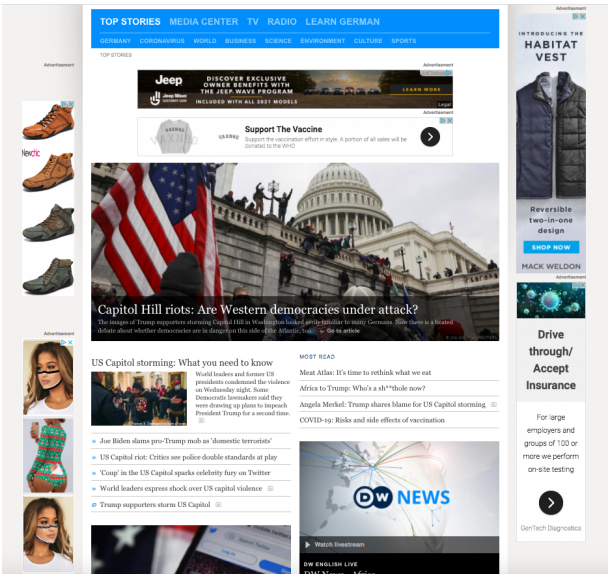


Figure 4.8: Snapshot of www.dw.com after cloning ads.

Instrumenting real-world websites is comparatively more arduous than synthetic pages due to the complexity and obfuscation of page sources. However, we adhere to the former for the sake of proximity to in-the-wild ads and fairness in our validations. Moreover, we duplicate every ad element (including leaderboards, infeed ads, sticky and animated banners, etc.) and do not limit ourselves to a specific type of ad for completeness. Without bias, we randomly pick *five* websites from our test corpus (see Section 4.5) and instrument them. Figures 4.7 and 4.8 show an example of this instrumentation on the appearance of a website where we observe that ads are duplicated. Note that ads are typically delivered from a bidding system (i.e., ad exchange) thus, a duplicated ad is not necessarily identical to the original ad. However, to minimize the impact of this stochastic behavior, we load websites multiple times and include cases where two ads (original and cloned) have at least the same structure

and size.

Table 4.1: Comparison of adPerf with Chrome DevTools in measuring the total computation time and breakdown by browser stages on a MacBook Air laptop.

Website		Total		Parsing		Scripting		Rendering		Painting	
		Time (sec)	Err. (%)	Time (sec)	Err. (%)	Time (sec)	Err. (%)	Time (sec)	Err. (%)	Time (sec)	Err. (%)
newindian express.com	adPerf	18.2		1.06		14.9		1.91		0.28	
	Chrome	18.6	**1.6**	1.17	**8.9**	15.3	**2.2**	1.79	**6.9**	0.26	**6.5**
buffalonews .com	adPerf	12.2		0.65		9.33		1.91		0.36	
	Chrome	12.6	**2.9**	0.58	**10.9**	9.88	**5.5**	1.83	**4.5**	0.33	**12**
huffpost.com	adPerf	2.13		0.13		1.61		0.30		0.09	
	Chrome	2.28	**6.8**	0.13	**<1**	1.76	**8.7**	0.31	**2.3**	0.08	**6.0**
observer.com	adPerf	3.12		0.16		2.40		0.47		0.09	
	Chrome	3.43	**0.41**	0.16	**0.43**	2.72	**0.49**	0.46	**0.57**	0.08	**0.57**
dw.com	adPerf	3.20		0.25		2.34		0.51		0.10	
	Chrome	3.38	**5.1**	0.24	**6.3**	2.54	**8.0**	0.50	**1.2**	0.09	**10.6**

Table 4.1 summarizes the results from the validation tests on the accuracy of adPerf in measuring the page loading workload and fine-grained breakdown by browser stages on the original webpage. We observe that adPerf measures total page-dependent browser computation within 0.4% to 6.8% of Chrome DevTools for the five randomly sampled test webpages. Besides, adPerf's breakdown is well in line with Chrome DevTools, and all the stage measurements are below a 12% margin of error, with a median error of 5%. This verifies adPerf's parser, pruning, call-stack construction, and activity classification are functioning accurately.

In Table 4.2, we present our results from the performance dissection of ads by both adPerf and Chrome DevTools for the same websites. For each website, *adPerf* indicates ads performance cost reported by adPerf on the original page, and *Chrome* signifies the ads cost estimated by Chrome DevTools (which is measured by calculating the difference in timings between the original page and the page with duplicated ads). The results show that adPerf's total ad costs are within 11% of Chrome DevTools estimation which confirms that adPerf's graph builder, call-stack analyzer, activity tracker, and resource matcher modules are designed correctly, and adPerf successfully isolates ads from the main content. Additionally, using adPerf, we measure the increase in the total page workload after cloning ads which are

Table 4.2: Comparison of the ads performance cost reported by adPerf with Chrome DevTools estimation on a MacBook Air laptop. *adPerf* indicates the ads performance cost reported by adPerf on the original page. *Chrome* indicates the ads cost estimated using Chrome DevTools by computing the difference in timing between the original page and the page with cloned ads. *adPerf 2x* indicates the increase in ad workload reported by adPerf after cloning ads.

Website		Total		Parsing		Scripting		Rendering		Painting	
		Time (ms)	Err. (%)	Time (ms)	Err. (%)	Time (ms)	Err. (%)	Time (ms)	Err. (%)	Time (ms)	Err. (%)
newindian express.com	adPerf	2750		134		2470		131		22	
	Chrome	2560	**7.1**	120	**10.4**	2270	**7.9**	142	**12**	25	**3.1**
	adPerf 2x	2840	**3.1**	146	**8.2**	2540	**2.9**	134	**4.5**	21	**7.1**
buffalonews .com	adPerf	635		91		411		121		12	
	Chrome	683	**7.0**	84	**7.7**	478	**14**	111	**8.3**	10	**23**
	adPerf 2x	520	**15**	81	**11**	349	**15**	94	**22**	16	**19**
huffpost.com	adPerf	694		56		432		163		43	
	Chrome	671	**3.3**	54	**3.6**	413	**4.4**	157	**3.7**	47	**8.5**
	adPerf 2x	730	**4.9**	59	**5.0**	458	**5.7**	168	**3.0**	45	**4.4**
observer.com	adPerf	400		17		258		117		8	
	Chrome	451	**11**	19	**10.5**	316	**18.4**	107	**8.5**	9	**0.41**
	adPerf 2x	352	**12**	16	**5.9**	225	**12.8**	104	**11.1**	7	**12.5**
dw.com	adPerf	1040		85		721		196		38	
	Chrome	971	**6.6**	93	**8.6**	641	**11.1**	190	**3.1**	44	**13.6**
	adPerf 2x	870	**16**	85	**<1**	684	**5.1**	172	**12.2**	44	**13.6**

denoted by *adPerf 2x* in Table 4.2. We then compare the former against the cost of the original ads for each website. This comparison shows that duplicating ads does not precisely double the performance cost of ads but is within an acceptable range of 3.1% to 16% of the original ads cost. The inaccuracy primarily stems from the fact that the cloned ads do not exactly resemble the original ads as seen from Figures 4.7 and 4.8 due to the bidding system. We attribute the marginal errors in the validation against Chrome DevTools to the same artifact and anticipate adPerf's reported ad cost to be even closer to reality.

4.5 Experimental Setup

System. Our system is a MacBook Pro with 2 cores and 8 GB RAM connected to a high-speed WiFi (400 Mbps). The mobile experiments are conducted on a Nexus 6P (quad-core ARM Cortex-A53 + quad-core ARM Cortex-A57 processor) connected to the cellular network. To obtain accurate results on communication overhead, we do not set up any proxy or local server.

Test corpus. Our test corpus consists of two sets of web pages – (a) top 350 websites from Alexa top 500 news list [11] and (b) top 200 websites from Alexa top 500 list [12]. We will refer to these two web page datasets as *news* and *general* respectively. The two lists have only 17 websites in common. For each dataset, we crawl the corresponding corpus twice. The first time, we crawl the home page or landing page of the website. The second time, we randomly click a link on the home page and crawl the page that it leads to. We exploit Chrome Popeteer [34] to automate link clicking. We refer to the former as the *landing* page crawl and the latter as the *post-click* page crawl.

Experimental repeat. In each crawl over the corpus (total 4 crawls), we load websites multiple times and take the average to account for fluctuations in page loading.

Evaluation domain. Since the main goal is to characterize the performance cost of ads, we primarily provide evaluation results for the websites that contain ads. This is nearly 80% of news websites and 40% of top general websites.

4.6 Results and Discussion

In this section, we analyze the performance cost of ads from two viewpoints – at the ad domains (close to the *origin*) and deeper in the browser (close to the *metal*). First, using

adPerf, we analyze the performance cost of ads on the websites broken down by costs incurred by the computation (i.e., rendering engine) and network (i.e., resource loader). Then, we investigate a level deeper to understand which computation stages and network resources mainly contribute to the computation and network ad costs respectively. Finally, we zoom out and analyze the ad domains themselves to quantify their contribution to the performance cost of web ads.

4.6.1 Computation Cost of Ads

For every website, we calculate the fraction of time spent in ad-related activities to the total activities (ad + non-ad). Figure 4.10 shows the CDF distribution of this fraction for the 4 different crawls.

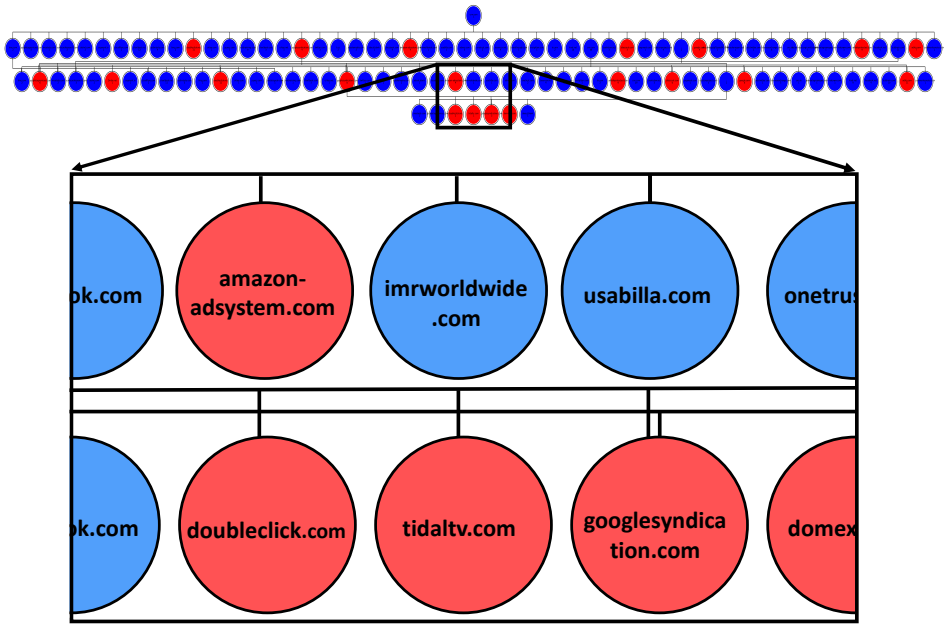


Figure 4.9: Resource-dependency graph for `www.cnn.com`. Ad nodes are colored red and non-ad nodes are colored blue.

Finding 1. According to the figure, web ads can have a significant impact on the performance of the website. For example, half of the news websites spend more than 15% of their computing time on ads. Moreover, 20% of the news websites spend more than 30% of the

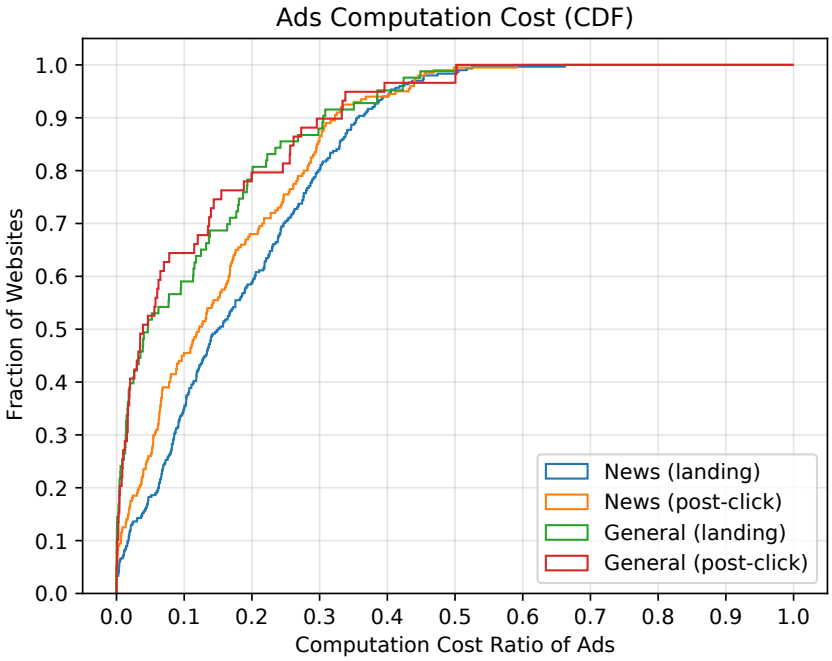


Figure 4.10: Computation cost of ads in two datasets namely top general and top news websites. Each domain in the dataset is crawled twice (landing page and post-click page).

time on advertising which can be concerning from the user's perspective. It also motivates website builders and ad providers to optimize their advert contents. Compared to the news websites, ads have a lower cost on the general corpus. The median in this corpus is 5%.

Finding 2. The figure presents another interesting detail when we compare the landing and post-click page graphs. Ads have a higher performance cost when loading the landing page versus the post-click page of news websites by about 25% on average. However, this is not the case for general websites. Post-click pages of popular general websites have almost similar cost-performant ads as the landing page. Further, we aggregate the total time spent on ad-activities across all browser stages and compare that to the time spent on the main content. The average percentage of time spent on ads versus main content for the news landing page, news post-click, general landing page, and general post-click datasets is 17, 15, 11, and 10% respectively. The averages are higher than the median percentages reported earlier because a small number of websites spend 40-50% of the computation time on ad-activities.

Breakdown of ad computation cost. Since we observe that ads can have a significant impact on website loading, it is worthwhile to explore the cause of this overhead. This can guide website builders and ad providers to focus their optimization efforts on those activities that are the primary sources of performance loss. Accordingly, we classify the computation cost of ads by the granularity of the browser stages (outlined in Section 4.2). Figure 4.11 shows the contribution of the six major stages for the news corpus. For each stage, s, we measure the following three metrics. Note that ct^s is the computation time of stage s while ct^* is the total time spent in computation across all the stages. Similarly, ct_{ad} is the computation time spent on ad-activities while ct_* is the total time spent on all activities. Therefore, ct^*_* is the total time of all computation activities in the browser.

1. The fraction of time spent on ad-activities in stage s to the total time spent on all activities in stage s $[ct^s_{ad}/ct^s_*]$. This is shown by the green bars. This *intra-stage* metric indicates how the workload of the stage, s, is split between ads and the main content.

2. The fraction of time spent on ad-activities in stage s to the total time spent on ad-activities across all stages $[ct^s_{ad}/ct^*_{ad}]$. This is shown by the blue bars. This *inter-stage* highlights how a particular stage, s, is impacted by ads compared to the other browser stages.

3. The fraction of time spent on all activities in stage s to the total page load computation time $[ct^s_*/ct^*_*]$. This is another *inter-stage* metric shown by the red bars. However, unlike the above metric, it shows the influence of a particular stage, s, on the entire page load.

It is important to correlate both the inter-stage metrics to have a complete analysis. For example, if a stage has a significant contribution to ads (i.e., second metric) but has very little impact on page loading (i.e., third metric), then it is unlikely to be a performance optimization target.

Finding 3. Figure 4.11 shows that *scripting* has the highest impact, more than 88%, on the computation cost of ads. It also has a significant impact (73%) on the computation workload of the entire page. The difference between these two metrics indicates that ads are more scripting heavy than the main content. This is because ad-content has 21% more dynamic characteristics than the original page content in our news corpus which increases the time spent in scripting. However, scripting only spends 25% of its time on ad-related content (i.e., first metric). Therefore, ads are not the primary bottleneck of the scripting stage but optimizing this stage will considerably improve the performance of ads as scripting is the major workload of today's web ads on news sites.

Finding 4. Another observation from Figure 4.11 is that HTML parsing has a minor influence on page loading, i.e., less than 5% in comparison with scripting but ads have more impact on this stage (comparing green bars). In other words, optimizing ads HTML code is expected to improve HTML parsing workload more than optimizing ads JavaScripts for the scripting stage, even though HTML optimizations can only marginally improve page load time. This underscores the importance of correlating the intra- and inter-stage metrics to guide optimization efforts. We observe similar behavior for the general corpus as well.

4.6.2 Network Cost of Ads

Besides computation activities, loading ads imposes overhead on the network activities. To measure the performance cost of ads over the network, for each website, we calculate the ratio of time spent on fetching ad-related resources to the total time spent on fetching all the requested resources. Figure 4.12 shows the CDF of this network cost ratio for the 4 crawls.

Finding 5. The four distributions follow the same order as in Figure 4.10 (computation cost

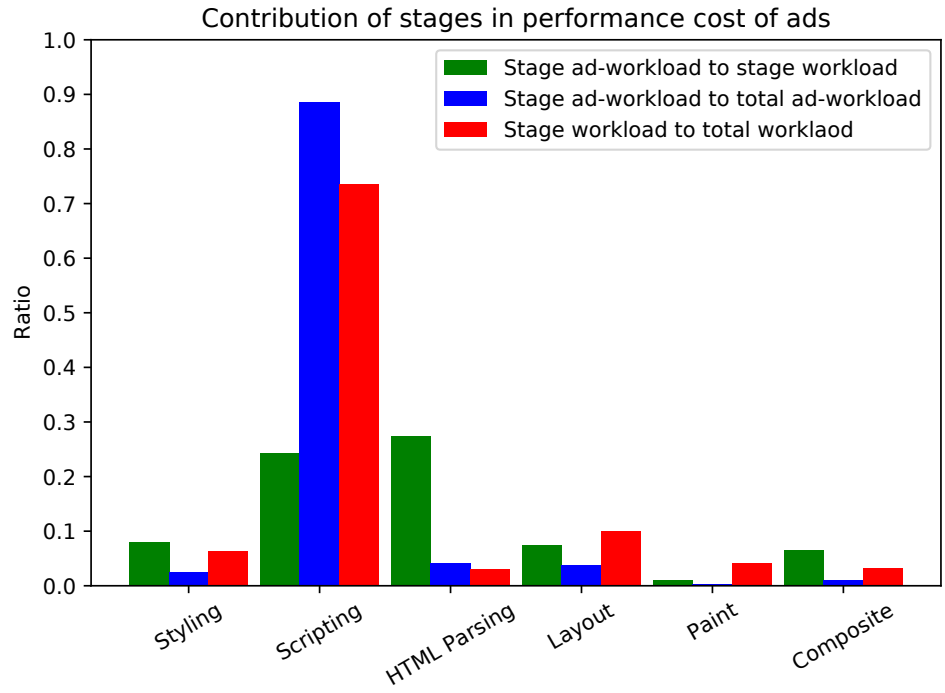


Figure 4.11: Contribution of the different browser stages to the performance cost of ads for the news landing corpus. The three bars for each stage correspond to the three ratio metrics (ct^s_{ad}/ct^s_*, ct^s_{ad}/ct^*_{ad}, and ct^s_*/ct^*_*).

of ads), i.e., news websites incur higher network performance cost than general websites. This is not surprising since more and/or larger ad resources also require more work in parsing, evaluating, and rendering. According to the figure, the median of the network-cost ratio is 15% for news websites' landing page and 3% less on the post-click page. For the general websites, the median is 6% for the landing page and post-click page respectively.

Breakdown of ad network cost. To dissect the network costs of ads, we breakdown the network time consumption by content type (such as HTML, image, and media). For each content type, Table 4.3 summarizes statistics about the frequency of resources and network time spent on fetching those resources for the news corpus for both landing and post-click pages. Given the number of resources, nr, and network time spent on the resources, nt, we define three metrics for each (similar to computation stages) as follows.

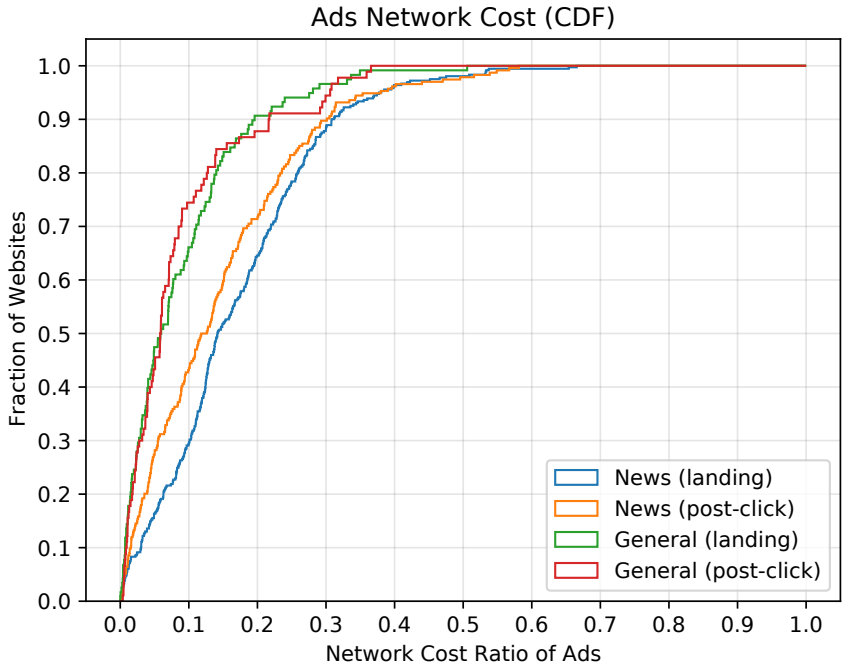


Figure 4.12: Network performance cost of ads in two corpuses: general and news websites. Each corpus contains landing and post-click pages.

Table 4.3: Summary of the three metrics each for the number of resources and network time spent on resources across two types of pages (landing page denoted by L and post-click page denoted by PC) for the news corpus.

content type	number of resources statistics (%)						request time of resources statistics (%)					
	nr^c_{ad}/nr^c_*		nr^c_{ad}/nr^*_{ad}		nr^c_*/nr^*_*		nt^c_{ad}/nt^c_*		nt^c_{ad}/nt^*_{ad}		nt^c_*/nt^*_*	
	L	PC	L	PC	L	PC	L	PC	L	PC	L	PC
Script	23	22	**41**	**45**	**40**	**43**	25	24	**49**	**57**	33	37
HTML	**36**	**34**	9	9	5	6	**17**	**14**	4	5	4	5
Image	23	22	**37**	**35**	**37**	**33**	13	12	**39**	**32**	50	42
Font	13	6	1	1	2	2	6	3	1	<1	2	2
CSS	6	3	1	1	5	6	5	2	1	1	3	4
XML	**54**	**46**	1	<1	<1	<1	**68**	**43**	**1**	**<1**	<1	<1
XHR	18	12	4	3	6	5	12	7	**5**	**4**	**7**	**8**
Media	4	4	<1	<1	<1	<1	3	3	<1	<1	<1	<1
Unknown	24	30	5	5	5	4	6	13	<1	1	1	1

- Metrics for the number of resources (nr).

    1. The fraction of the number of resources of content type, c to the total number of resources of c $[nr^c_{ad}/nr^c_*]$ (intra resource-type metric).

    2. The fraction of the number of ad-resources of content type, c to the total number of ad-resources (of all content types)$[nr^c_{ad}/nr^*_{ad}]$.

    3. The fraction of the number of resources of content type, c to the total number of resources $[nr^c_*/nr^*_*]$ to highlight the popularity of the content type.

- Metrics for the network time spent on resources (nt).

    1. The fraction of the network time spent on ad-resources of content type, c to the total network time spent on resources of c $[nt^c_{ad}/nt^c_*]$.

    2. The fraction of the network time spent on ad-resources of content type, c to the total network time spent on ad-resources (of all content types) $[nt^c_{ad}/nt^*_{ad}]$.

    3. The fraction of the network time spent on resources of content type, c to the total network time spent on all resources $[nt^c_*/nt^*_*]$ to accent the performance impact of content type, c.

For instance, the first metric for network time of CSS refers to the fraction of time spent on fetching ad-related CSS resources to the time spent on fetching all CSS resources $[nt^{css}_{ad}/nt^{css}_*]$.

Finding 6. Among all content types, Table 4.3 shows that XML has the largest percentage of ad resources for both landing (54% which account for 68% of the network time in fetching XML resources from metric 1) and post-click pages (46% which take up 43% of the network time). However, it contributes to an insignificant fraction of the network performance cost for both pages (metric 2). On the contrary, scripts and images commonly used by ad providers, make up nearly 80% of all ad resources (metric 2) *and* all resources (metric 3) for both landing and post-click pages. Among the two content types, scripts on average are 20%

more popular than images for post-click pages compared to the landing page (comparing metrics 2 and 3). Script files used in advertising alone are responsible for almost half of the network performance cost of ads, followed by images at 40% for landing pages (metric 2). More scripts in post-click pages correspond to a higher contribution to the network time spent in ads (57%) for these pages compared to images (33%).

Finding 7. Ad-related HTML files constitute 34-36% of total HTML files but they only take 14-17% of download time. A deeper investigation shows that ad HTML documents are lighter than main-content HTML. The former has a significantly small number of tags (on average 7) including only one or two <script> tags that encapsulate small and minified code compared to the main-content HTML files with 410 tags. Surprisingly, XHR (XMLHttpRequest) resources make up a significant 7% of the network performance cost for the landing page and 9% for post-click pages (metric 3). The corresponding time spent on ad resources is 5% and 4% respectively (metric 2).

4.6.3 Breakdown of Ad Performance by Source

The results so far breakdown the performance cost of web ads at the lower level of granularity. Now, we zoom out and quantify the cost of ads based on their origin, i.e., ad domains. The goal of this lens is to gain an understanding of the third-party ad domains and their impact on the performance cost. Accordingly, we build the resource-dependency graph (as described in section 4.3) for all news websites in our test corpus. Overall we identify more than *300 distinct* ad domains.

Breakdown of computation performance cost by ad domains. For every ad domain, we first aggregate the time the rendering engine spends on evaluating the resources served by that domain. We also measure the total time spent on ads through the crawl (ads computation cost). The ratio between the above two is an indicator of how each third-party

ad domain contributes to the computation cost of ads. Figure 4.13 shows the contribution of the top 10 ad domains (out of 300) in decreasing order (from left to right) of their performance impact. The number on top of each bar is the number of websites in our corpus that are served by that ad domain.

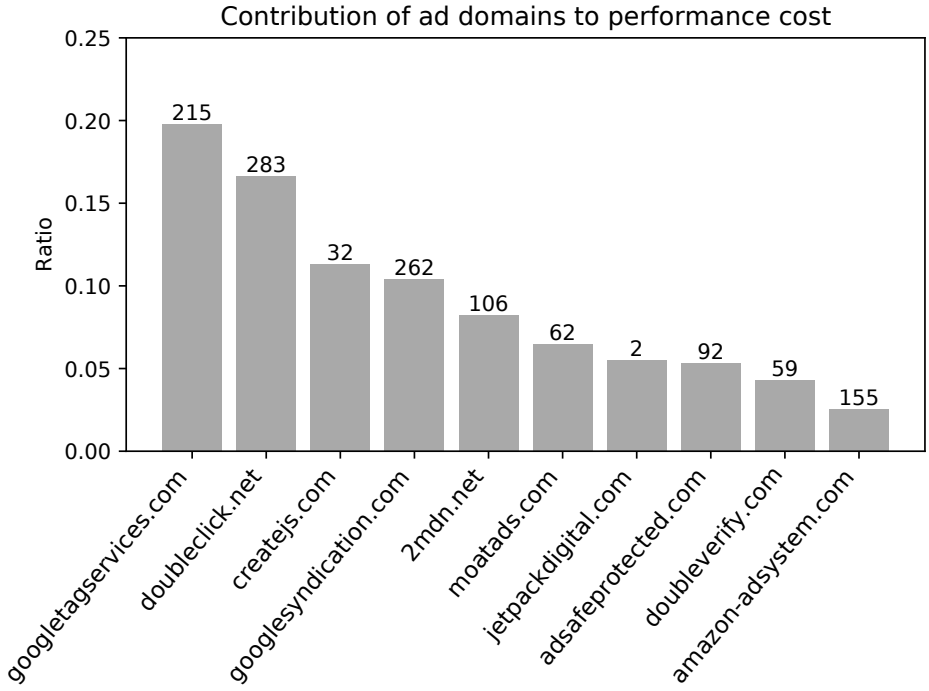


Figure 4.13: Contribution of ad domains to the computation cost of online advertising. The number on top of each bar is the number of websites serviced by that particular ad domain.

Finding 9. `googletagservices.com` and `doubleclick.net` have the highest contribution to the computation of ads on the web. The former is a Google tag management system for managing JavaScript and HTML tags used for tracking and analytics on websites, and the latter is a popular ad provider. Together, they deliver about 35% of the total ad resources. Moreover, all the ads are not delivered by well-known ad domains. In our corpus, 50% of ad domains appear only in the dependency graph of *one* website.

Besides, the number of websites serviced by an ad domain is not an indicator of its performance cost. For instance, `googlesyndication.com` has approximately the same contribution to the performance cost of ads as `createjs.com` but it services over 8× more websites than

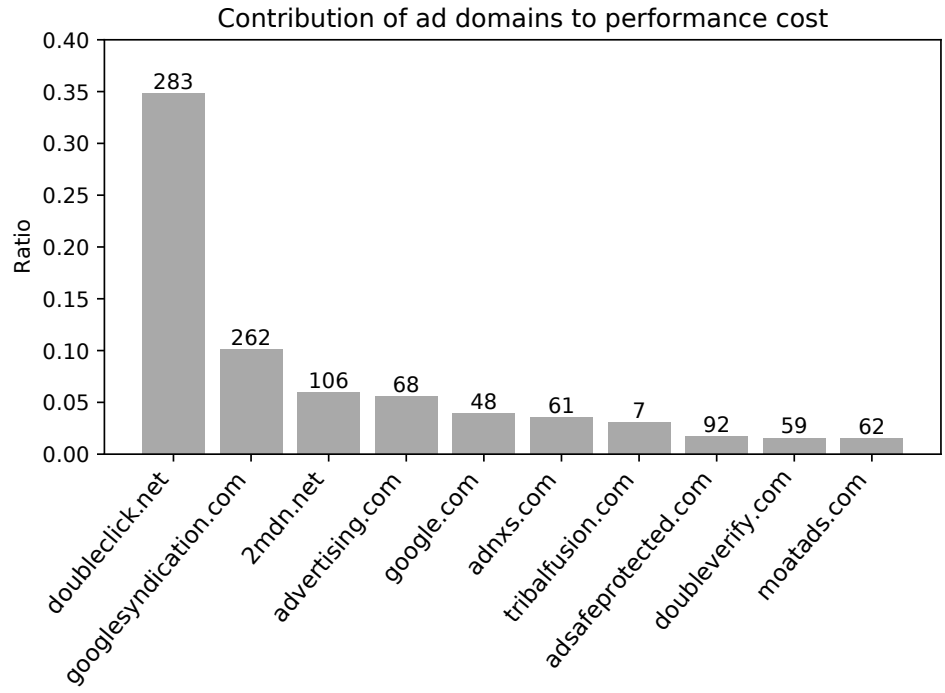


Figure 4.14: Contribution of ad domains to the network cost of online advertising.

the latter. This is because `createjs.com` provides content for interactive ads (flash-like ads using HTML5 canvas) that trigger JavaScript callback functions constantly to sporadically change the content and re-flow. `createJS` ads (where usually incorporated by intermediate ad domain) on 32 websites of our corpus heavily use Scripting activities, 7.5% more than Scripting activities belong to `googlesyndication.com` on 262 websites.

Breakdown of network cost by ad domains. We follow a similar procedure as above for estimating the contribution of individual ad domains to the network cost of a page load. For every ad domain, we first aggregate the time the browser spends on fetching resources by that domain, Then, we calculate the ratio of the total time spent on fetching ad resources in our crawl to the above time. Figure 4.14 shows the top 10 ad domains that have the highest contribution to the network cost of ads in the news corpus.

Finding 10. About 35% of the network cost of ads on news websites is traced to `doubleclick.net` followed by the popular ad syndication `googleadsyndication.com` with 10% contribution.

Google is **the major actor** in the ad ecosystem. Domains maintained by Google alone constitute approximately **51%** of the total ad network cost.

Finding 11. Comparing the computation cost of domains with their network cost shows that these two performance costs are correlated. As one might expect, fetching more and larger documents also takes longer to evaluate and display. Interestingly, we also observe domains that have a high computation cost but insignificant network cost and vice versa. For instance, `googletagservices.com` has the *highest* contribution (19.7%) to the computation cost of ads among all 300 ad domains. However, it contributes to less than 1% of the network cost (ranked 16 and not shown in the top 10 domains in Figure 4.14). Further breakdown of its performance cost with adPerf reveals two JavaScript documents (`osd.js` and `osd_listener.js`) of size less than 76 KB belonging to this domain referenced by over 200 websites in the news corpus. These two files are part of Google Ads that track the viewability of the ads to assess the value of an impression to the publisher and advertiser. To calculate what percentage of an ad appears in a viewable space on the screen and for how long that portion of the ad remains visible, these JavaScript snippets are frequently invoked by the webpage and take up valuable CPU cycles.

Breakdown of performance cost by trustworthiness. When a publisher displays an ad on their webpage, there is an *explicit* trust between the publisher and the provider. However, when the ad provider is part of a syndication, the ad is served through a chain of redirections going through different ad domains. Our measurement results on the Alexa news and general websites shows that the mean depth of this chain is **4**, revealing ad syndication is prevalent. Most of the ad domains on the chain are not directly visible to the publisher (except the ones directly embedded by the publisher). As a result, the publisher cannot verify their intention (e.g., whether used for drive-by download or phishing). This results in the publisher placing an *implicit* trust in the ads since the trustworthiness of these ad domains is unknown. In this work, we are interested in the correlation between the performance cost of an ad domain

and its trustworthiness.

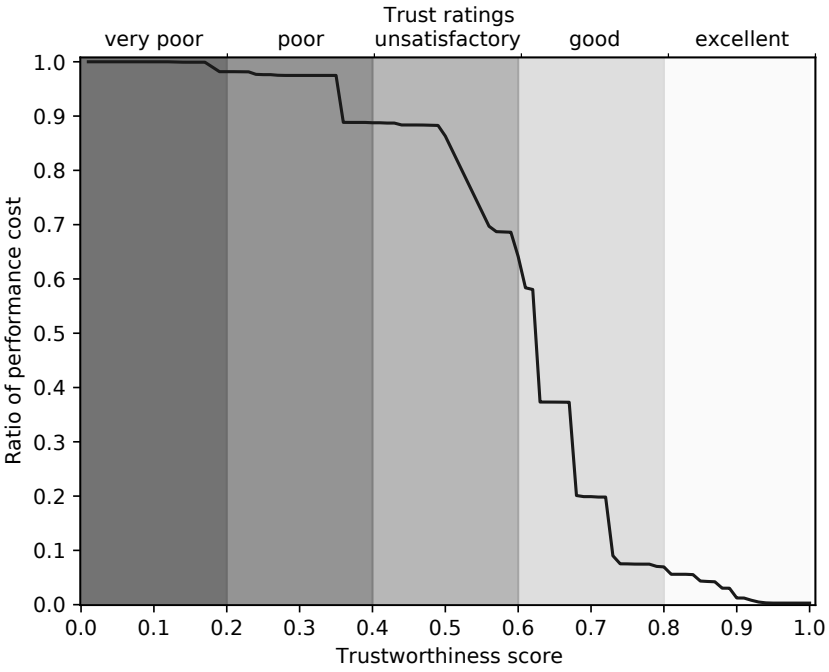


Figure 4.15: Performance cost of ads delivered by ad domains as a function of WOT trustworthiness score (CDF). Scores are normalized to [0,1] and different colors highlight different trustworthiness rating.

To this end, we leveraged two online services, WOT (Web of Trust) [39] and VirusTotal [38], to model the trustworthiness of an ad domain. WOT is a community-based reputation system that assigns a score to a domain name based on user complaints and other blacklists. The score ranges from 0 to 100, and WOT classifies domains based on their scores into 5 *trust rating* – excellent, good, unsatisfactory, poor, and very poor [76]. VirusTotal is a portal that proxies the request of a security check of a domain/URL to its affiliated blacklist services (71 blacklists).

When a domain is submitted to VirusTotal, it reports the blacklists that flag it as *red*. We count the ratio of blacklists that do not raise an alarm on the domain (i.e., safe flag) as the VirusTotal score (i.e., 0 means highly malicious and 1 is completely benign). Both WOT and VirusTotal have been used to determine the trustworthiness of a domain by previous

studies [71, 95, 76, 94].

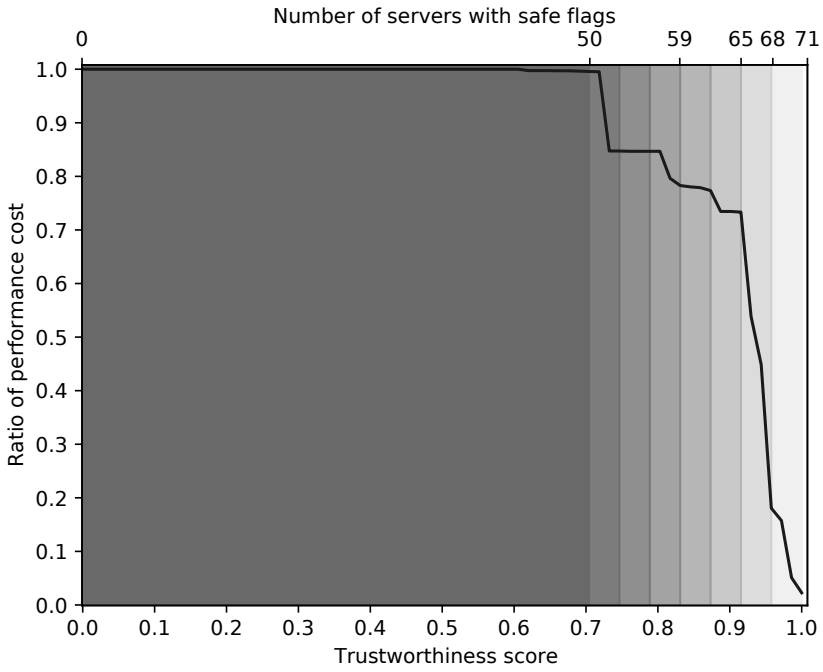


Figure 4.16: VirusTotal

Figure 4.17: Performance cost of ads delivered by ad domains as a function of VirusTotal trustworthiness score (CDF). Scores are normalized to [0,1] and different colors highlight different trustworthiness rating.

One challenge we faced is determining thresholds for trust ratings since it varies widely across different services that report a *trustworthiness score*. Therefore, to provide a fair analysis, we report the contribution of domains to the ad cost for different thresholds. Figures 4.15 and 4.17 illustrate the cumulative performance cost of ad domains as a function of trustworthiness assessed by WOT and VirusTotal, respectively. For WOT, we use its default classification (5 classes) [76]. For VirusTotal, we observe that almost all of the domains receive at least 50 safe flags, so we only breakdown the region from 50 to 71 servers at the granularity of 3 servers.

Finding 12. Following the default classification of WOT, about 63% of ads cost is from ads delivered by trusted ad domains (excellent and good rating). Nevertheless, domains that are not trusted (unsatisfactory, poor, and very poor rating) contribute to a considerable portion

of ads (37%) which is a flag for publishers. Accordingly, for VirusTotal, we see that only 5% of the performance cost of ads is from domains that don't receive any red flags.

Finding 13. Domains that are moderately trusted (i.e., neither highly trusted nor untrusted) have the highest contribution to the performance cost of ads as seen from Figures 4.15 and 4.17 . The amount of drop in the fraction of performance cost (y-axis) within each shaded region indicates the performance cost for that level of trust. For example, domains with more than 80% WOT score (excellent trust rating) contribute to 5% of ads performance cost while 58% of ads cost belongs to domains with 60% to 80% score (good trust rating). Likewise, domains with less than 3 VirusTotal red flags (first shaded region from the right) account for 18% of ads cost but 55% for domains with 3 to 6 red flags (second shaded region from the right). Our results do not assert a strong correlation between trustworthiness and the performance impact of third-party ad domains.

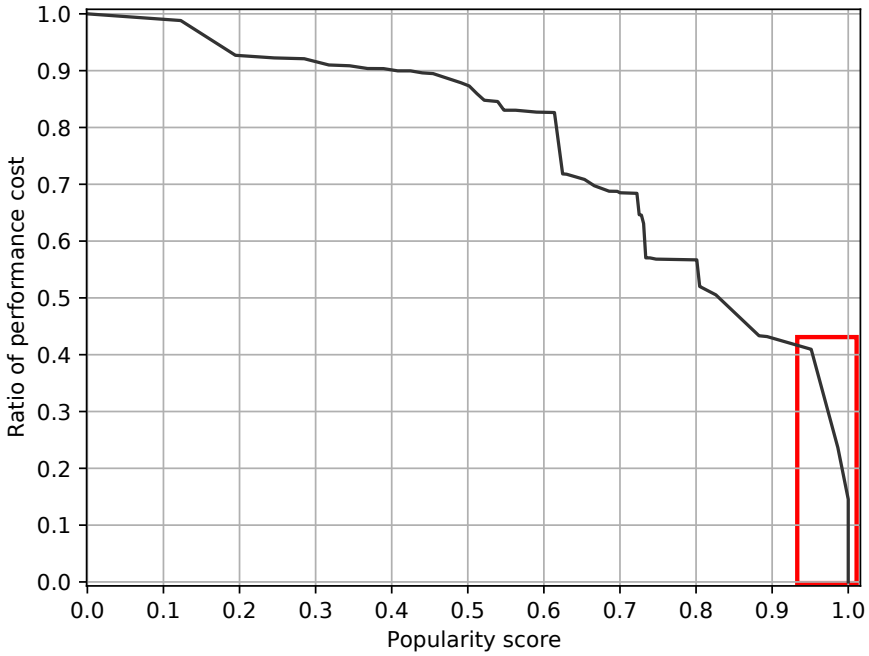


Figure 4.18: Performance cost of ads from popular domains as a function of popularity score (CDF) based on number of referrers.

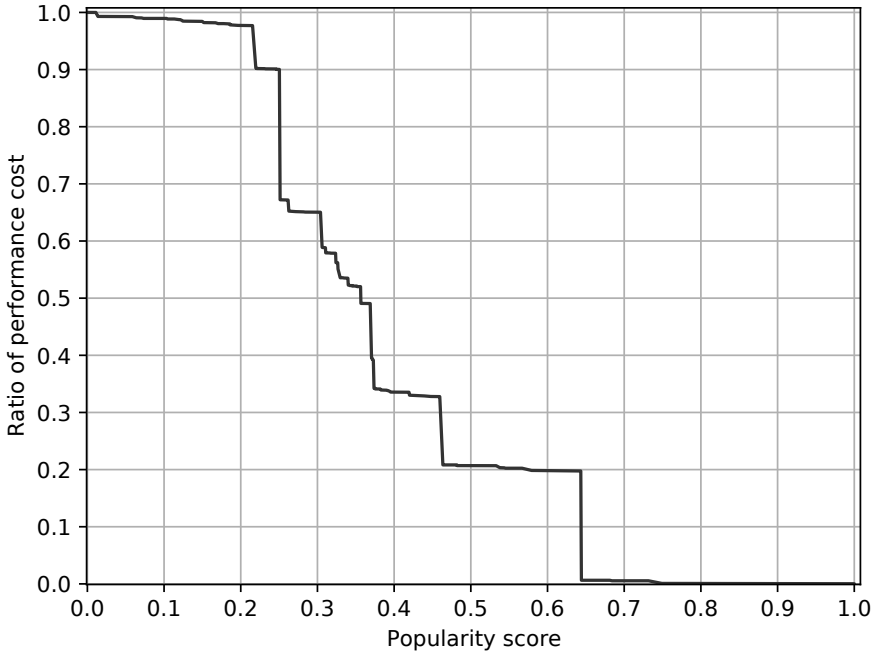


Figure 4.19: Performance cost of ads from popular domains as a function of popularity score (CDF) based on Alexa ranking.

Breakdown of performance cost by popularity. Similar to trustworthiness (gauged by the delivered content), we can study the relationship between the popularity of an ad domain and its performance impact. Accordingly, we model the domain reputation by its popularity, which is determined by the Alexa ranking [11], and the number of websites in our corpus to which it delivers ads. However, there is no agreed-upon cutoff to split ad domains into popular versus unpopular. For this reason, we follow a similar method to the trustworthiness study and present the performance cost of ad domains at varying cutoff levels. Figures 4.18 and 4.19 illustrate the cumulative contribution of popular domains to the performance cost of ads for two metrics.

Finding 14. Earlier in this section, we observe no correlation between the popularity of the ad domains (i.e., number of referred websites) and the performance cost for multiple domains. However, at the macro-level, more popular ad domains contribute more to the

performance cost as seen from Figure 4.18 and this is due to higher reach of those domains. As highlighted in this figure, the fraction of performance cost drops about 40% within a 5% range of the most popular ad domains. However, for the Alexa ranking (Figure 4.19), we observe multiple sharp drops throughout the score range, meaning there exist multiple ad-domains that have a significant contribution to the performance that is neither very popular nor very unpopular.

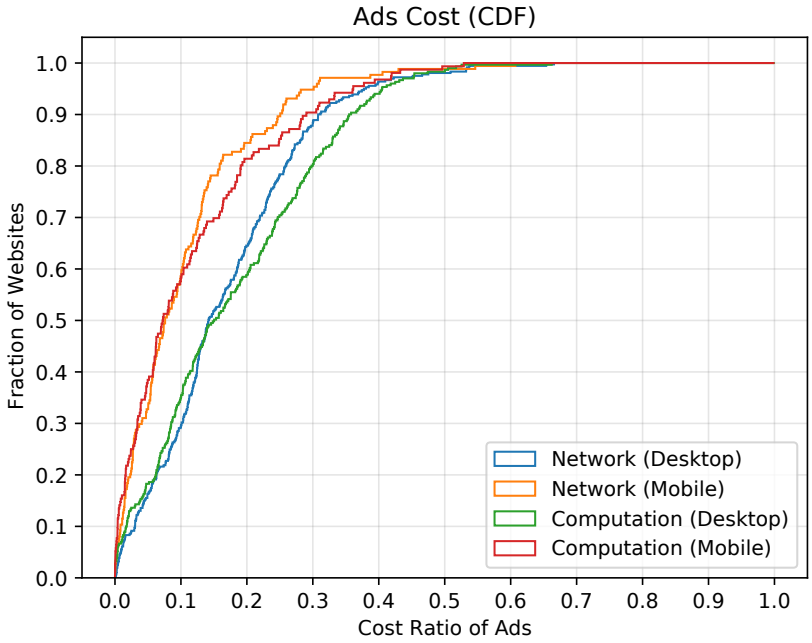


Figure 4.20: Comparison of performance cost of ads (computation and network) for mobile and laptop on news corpus.

4.6.4 Desktop vs. Mobile Ads

Mobile represents a significant medium for web consumption. We repeat the experiments and load webpages on the Nexus 6P smartphone to evaluate the performance cost of web ads on mobile devices using adPerf. adPerf uses port forwarding to connect to the device and Chrome remote interface to capture traces remotely on the device which they are then parsed and analyzed on the host system. Figure 4.20 shows a comparison of the performance

content type	nr^c_{ad}/nr^c_*
Script	17.5%
HTML	25.7%
Image	12.8%
Font	2.2%
CSS	2.3%
XML	36%
XHR	6.4%
Media	6.3%

Table 4.4: Fraction of ad documents to total documents for each content type (nr^c_{ad}/nr^c_*).

Domain	Cost
doubleclick.net	28%
googletagservices.com	20%
googlesyndication.com	18%
ampproject.com	7%
cloudfront.net	4%
2mdn.net	4%

Table 4.5: Top ad domains contribution to performance cost of mobile ads.

cost of web ads on mobile and desktop for the landing page of the News corpus.

Finding 15. Mobile ads add on average 8% overhead to the page loading computation and the same amount on the network consumption. This is a notable 7% less than desktop pages. Further breakdown of computation cost by browser activities show a similar contribution of stages to the ads workload, with *Scripting* being the highest amounting to 87%. However, the amount of scripting work spent on ads to the total scripting workload reduced from 25% on the laptop to 13% on mobile.

Our assessment shows that this performance gap is because websites display fewer ads and they are better optimized on mobile to deliver content on a smaller screen. Overall, the fraction of ad documents to the total documents in the news websites dropped from 22.5% to 15.5% on the mobile device. Table 4.4 breaks down the above fraction by each content type (nr^c_{ad}/nr^c_*) except Media for the mobile version. Across the board, a fewer number of ad documents are fetched in each category compared to results on desktop (Table 4.3). For instance, ad images and scripts dropped by 44% and 24% in the mobile crawl. A recurring pattern in the websites is the absence (or limited inclusion) of skyscraper ads on mobile with a marked difference in the performance cost. This trend is illustrated in Figure 4.21 for Deutsche Welle, the popular German-based international news broadcaster. In *m.dw.com* (the mobile version on the right), two side skyscraper display ads are replaced with one

in-feed ad, reducing 10% of ads performance overhead.

We investigate the sources of mobile ads and breakdown the performance cost by third-party domains. Table 4.5 shows the contribution of the top mobile ad domains.

Finding 16. The sources that deliver ad content on mobile are fairly different from the desktop version and they have a dissimilar contribution to the performance cost of web-ads. For example, about 7% of the performance of ads on mobile comes from `ampproject.com` that specifically provides optimized ads for AMP (Accelerated Mobile Pages). Similar to desktop, `doubleclick.net` and `googletagservices.com` have the highest contribution in mobile advertising with a combined 48% share of performance cost, 13% more compared to desktop.

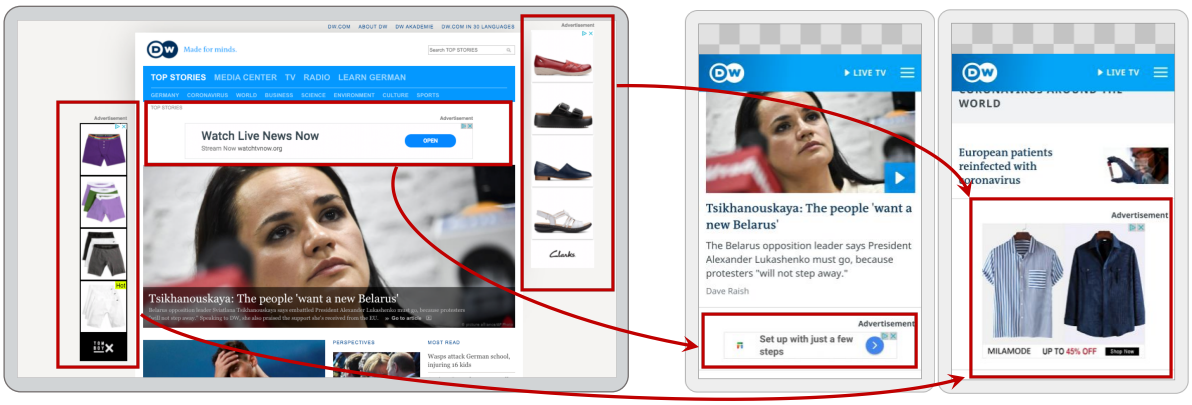


Figure 4.21: Snapshot of Deutsche Welle website on laptop (left) and mobile (right). Two side skyscrapper ads are substituted with one in-feed ad on the mobile.

4.6.5 Applications

AdPerf and the measurement study from different viewpoints present multiple applications for web researchers, browser developers, ad designers, content publishers, and perhaps even users. Notwithstanding, we discuss two use-cases below.

Heavy-ad intervention. Ads that consume a disproportionate amount of resources such

as draining the battery or eating up bandwidth on a device negatively impact the user experience. Intrusive ads range from the actively malicious, such as crypto-miners, to benign content with inadvertent bugs or performance issues. Chrome is launching an extension to limit the resources an ad may use and unloading that ad if the limits are exceeded. Tentatively, they define the following criteria and coarse-grained thresholds to limit the ads [26].

- Uses the main thread (i.e., the renderer thread which executes the majority of the computation activities) for more than 60 seconds in total or more than 15 seconds in any 30-second window.
- Uses more than 4 megabytes of network bandwidth.

The above metrics have been reported to have blocked many non-intrusive ads by Google Ad Manager Native Video and YouTube Skippable Preroll ads [27, 19]. AdPerf can aid Google engineers and potentially other browser developers to extensively study and characterize the performance of intrusive heavy-ads in different computation stages and network resources to establish a fine-grained threshold and criteria.

High-performance ads. AdPerf provides insight and guidance to both publishers and third-party ad providers to improve the performance of ads by identifying the stage and/or resource which are the main bottlenecks. For example, if adPerf identifies Scripting to be the computation and network bottleneck of ads on a website, one can follow targeted optimizations to loading third-party JavaScript such as lazy-loading scripts and libraries (e.g., serving an ad in the footer only when a user scrolls down the page), splitting JavaScript bundles (e.g., dynamic import() statement), self-hosting scripts with Service workers particularly for ad domains with consistent APIs, using resource hints like preconnect and DNS-prefetch, sandboxing script with iframes, using asynchronous ad tag manager in the code, and other

cognate recommendations provided by Google Page Insights [31] and Lighthouse [28]. Likewise, if Painting turns out to have excessive computation overhead for ads, this is likely due to animated GIF in the background of ad iframe or animation triggered by CSS (e.g., @keyframe rules). Ad designers can follow the recommendations on painting optimization, such as limiting manipulation to *transform* and *opacity* CSS properties that avoid repainting. Our analysis on ad domains can also advise publishers to select their ad providers from reliable syndications (i.e., with satisfactory trustworthiness score) that at the same time have a minimal performance impact, considering their reach.

4.7 Related Work

Over the past years, there have been a handful of studies on the performance characterization of web browsers and online advertising. Prior research mainly uses adblocker and adheres to page load time as the performance metric for characterizing the computation cost of ads; i.e., compare PLT before and after content blocking to measure ads cost. On the other hand, adPerf uses the ratio of the ad workload to the main content workload. Although adPerf's metric provides additional insight into the computation cost of ads, it cannot be directly compared to PLT for two reasons: (i) PLT combines both network and computation cost into a single metric and (ii) parallelization among activities in the browser. Moreover, with adblocking, we cannot decompose the performance cost into lower-level browser stages (e.g., HTML parsing and JavaScript) since they block resources at network initiation, and subsequent resource parsing, evaluation, and rendering are not captured. Therefore, in this section, we quantitatively and/or illustratively compare related work against adPerf.

4.7.1 Performance Analysis of Ads

Garimella et al. [92] analyze the performance efficiency and network overhead of popular ad blockers such as Adblock Plus [9], Ghostery [23], uBlock [37], etc. According to their data, blocking ads with Adblock Plus (Easylist rules) saves roughly 34% on cumulative network request time. This is higher than our measured network cost (15%) without the deployment of an adblocker. Although their setup is different and their corpus of news websites is small (Alexa top 150), they observe an increase in the number of network requests. This is due to various tracking services of their own and request for JavaScript modules designed for counter ad-blocking. Besides, they report a 15% to 43% increase in the CPU wall-clock when they use ad-blockers (Adblock, Adblock Plus, ublock, and Privacy Badger) and conclude that the time to load pages is not necessarily faster due to the overhead of ad blockers.

Butkiewicz et al. [73] break down the content of non-origin requests by MIME type and reports images and HTML/XML contribute to 42% and 9% respectively, which is slightly higher than our measurements, whereas, JavaScript contribution (25%) is far less than our measurements. Given the fact that 70% of these non-origin requests belong to advertising and analytics, this comparison signifies the rise of responsive and interactive ads within the past few years. Additionally, they attempt to quantify the cost of third-party content on page load time. By blocking non-origin content (using custom adblock filter), they measure 25% contribution. However, they report a 15% contribution when they consider the impact of non-origin requests on wall-clock rather than content blocking. Although their latter method does not preclude the content blocking overhead, it dismisses the parallelization among network requests and browser rendering activities associated with resources, hence is not reliable.

In other related studies [128, 151], authors deploy ad blockers in the wild and then use passive measurements on the traces to characterize the network traffic. Both studies report

17-18% of the network requests to belong to adverts, which is close to our numbers. Similar to our measurements, in [128], network requests are broken down by content type. However, the authors do not completely isolate the content types (e.g., CSS and JavaScript are not categorized), and therefore, a direct comparison is not feasible. Nevertheless, none of the studies investigate the effect of ads on the computation cost of page loading.

4.8 Conclusions and Takeaways

Our evaluations on the performance cost of ads lead to multiple new and interesting observations. The key finding of this research is that ads have a significant cost, more than 15% of the computation workload. This cost is relatively less in mobile browsing due to fewer and optimized ads for a smaller screen. Moreover, we discover Scripting contributes to $\approx 88\%$ of this cost in both environments suggesting ad designers to focus more on optimizing their JavaScript codes and publishers to follow practices for lazy loading of these scripts. We also find that ads have a different fingerprint on browser activities and web documents compared to the main content. For example, HTML parsing takes up only 5% of browser page loading workload (the lowest among other stages) but 29% of that is spent on ad-related content, more than any other stage, and XML files are requested more by the ad contents compared to the primary contents. Practitioners can use this anomaly to build a system for detection and intervention of ads or a subset of them (e.g., intrusive ads). Our evaluation also shows that a considerable fraction of the performance cost of ads is from untrusted domains which is a signal for the web community and to publishers to reconsider their ad-delivery network.

In this study, we did not account for ad resources that might be directly embedded in native HTML and ad resources that cannot be detected by filter lists (i.e., websites that use circumvention to evade filter lists). In future work, we plan to also include such sources of ad content. This addition would only increase the performance costs of different ad breakdowns

reported throughout this chapter, which we believe is already significant enough to warrant deeper attention. This work primarily aimed at designing a methodology and open-source infrastructure for fine-grained analysis of ads which we anticipate to be a useful tool for web researchers to prioritize their optimization efforts on web ads and publishers to analyze the impact of ads on their websites.

Chapter 5

Profile-based Tailor for Deep Learning Models

Running multiple tasks concurrently is a popular optimization technique for masking the program's latency and enhancing the end-to-end throughput. Finding the salient spot for exploiting concurrency is non-trivial because certain conditions such as tasks independence and resource availability must be met. A PA application that we examine in this chapter is deep learning model training on a GPU. Despite the fact that model-parallel and data-parallel schemes exist for distributed GPU training, single node GPU typically hosts a single-stream (i.e., serial execution) of neural network layers. Here we present a systematic profile-based PA and show the benefit of intra-layer parallelism to shorten training time. In cases where the default configuration of layers does not meet the conditions for parallel execution (e.g., resource constraints), our performance analyzer tailors the operations and determines the optimal configuration to accommodate concurrent network layer execution.

5.1 Introduction

Deep learning (DL) models have been rapidly growing and evolving within the past few years, with a wide range of applications. Take Convolutional Neural Network (CNN), a popular class of Deep Neural Network (DNN), as an example. Many CNN models have been developed in the last decade for learning problems in applications such as computer vision [105, 135], voice recognition [62], recommender systems [154], natural language processing [102, 79, 121], physics simulations and biosensors [119, 144, 142, 141, 87, 120]. Earlier CNNs were composed of a *linear* sequence of dependent layers like VGG and AlexNet. However, modern networks such as ResNet, GoogleNet, DenseNet, and PathNet have a more complex architecture. These *non-linear* networks [145] contain multiple fork/joins resulting in independent paths of chained operations. Figure 5.1 illustrates the difference in structure between linear (AlexNet) and non-linear (GoogleNet) networks.

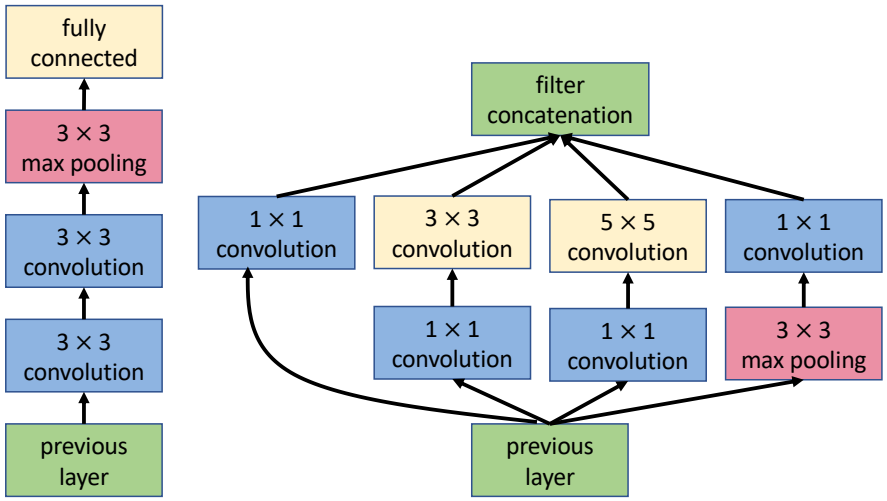


Figure 5.1: Linear (AlexNet on left) vs non-linear (GoogleNet on right) network.

GPUs are the platform of choice for training DL models. Training large-scale DNNs is extremely time-consuming due to the ever-growing number of parameters that have to be learned and the numerous iterations for the model to converge. Two approaches to reducing training time are to increase throughput and reduce the per-iteration execution time. For

the former, it is common to parallelize iterations across multiple GPUs. For the latter, however, there are several solutions in literature [140]. They can be broadly classified into either optimizing the operations in each layer or exploiting the concurrency between CPUs and GPUs by pipelining initial pre-processing operations (such as resizing, normalization) on the CPU with the rest of the operations on the GPU [145, 137]. As one can infer from Figure 5.1, unlike linear networks, non-linear networks have multiple independent operations across layers. However, none of the above existing approaches exploit this parallelism across multiple paths by running independent operations across layers concurrently on a single GPU. In this chapter, we investigate *why and how* to utilize this rich inter-op parallelism in non-linear DNNs (particularly CNNs) to reduce training time.

5.2 GPU Architecture and Programming Model

GPUs were originally designed for rendering images and graphics pipelines but they soon became a compelling platform for high-performance computing and machine learning due to their high peak performance and memory bandwidth. Subsequently, applications started reusing graphics API such as OpenGL and DirectX for parallel processing [113, 156, 132]. NVIDIA later released CUDA [57] as a standard programming model for scientific computing for their GPUs. AMD, another giant GPU manufacturer, uses HIP [58] programming language for their GPUs that is very similar to the CUDA. Figure 5.2 shows the GPU architecture and programming model.

In CUDA (or HIP), programs are typically divided into two parts – (1) *host* functions in the mainstream that is handled by the CPU threads and run on CPU cores and (2) *device* functions (aka. *kernels*) that are managed by GPU *streams* and executed on the GPU cores (device)[125]. CUDA organizes device code using abstractions of threads, blocks, and grids. *Kernels*, functions inside device code, are executed by CUDA *threads* in parallel, each on

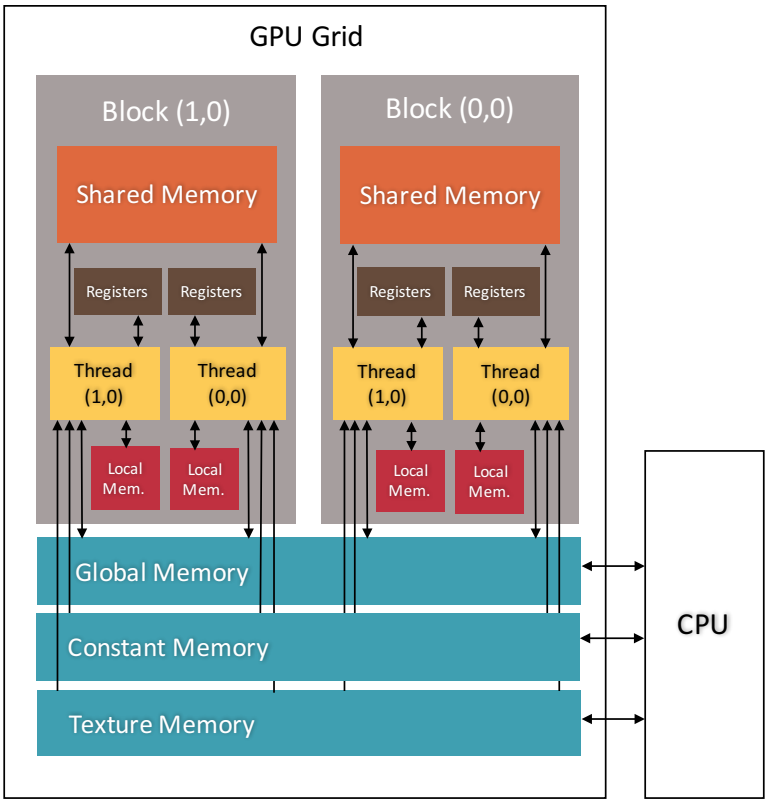


Figure 5.2: GPU architecture and programming model

a single GPU core. *Grid* is the entire kernel domain that is partitioned into *blocks* that include groups of cooperating *threads*. While threads inside a block can be synchronized, they execute their own instructions on a separate GPU core. CUDA assigns each block to one GPU processing unit which consists of hundreds of GPU cores. This unit is called *Streaming Multiprocessor (SM)* in CUDA and *Compute Unit (CU)* in HIP. There is also a hierarchy of memories in GPU with different access policies as shown in the figure. *Global memory* is accessible by the entire grid (all blocks and threads within a block), *shared memory* is local to each SM (i.e., only visible to threads inside the resident block), and *registers* are thread-local where each thread has a limited number of them. *Local memory* is an extension of registers that resides in global memory. GPUs also have other types of memories known as *texture* and *constant* that have the same access policy as the global memory but they are read-only.

5.3 Parallel DL Operations on GPUs

A majority of DL frameworks have a GPU backend that compiles the model and generates a computation graph at the granularity of basic operations such as convolution, batch normalization, and pooling. The operations are executed on the device by calling the corresponding APIs in *highly optimized* third-party libraries such as Nvidia cuDNN [75] and cuBLAS [1]. The kernels implemented in these libraries hold device resources to perform the DNN operations.

Layer	Algorithm	Kernel name	Register	Shared Mem	Thread	Block	ALU	Mem stall
L4-1	PRECOMP_GEMM	implicit_convolve_sgemm	92%	39%	38%	19%	70%	0.47%
$(3*3)$	FFT_TILING	fft2d_c2r_32x32	38%	75%	25%	6%	30%	15.2%
L4-2	PRECOMP_GEMM	implicit_convolve_sgemm	100%	70%	50%	100%	60%	0.03%
$(5*5)$	FFT_TILING	fft2d_c2r_32x32	38%	75%	25%	6%	20%	16.5%

Table 5.1: Resource utilization of two different algorithms for two independent convolutions in 4th layer of GoogleNet.

To launch multiple operations concurrently on a GPU, each operation has to be assigned to a separate executor (*stream* in the CUDA programming model). Besides, to accommodate two or more operations on a GPU, DL frameworks need to ensure there is enough device memory available at launch time [1]. To locate an opportunity for parallelization among DL operations, we propose a concurrency-aware tailor for DL operations on GPU (or simply catDog). Basically, catDog is a systematic PA technique that uses profilers under the hood. It profiles all network operations in the first iteration and based on the profiling results suggests the possibility of concurrency and performance gain, even though such parallelism is hard to implement. In the next section, we take CNN as an example and demonstrate our methodology and challenges along with supporting preliminary results.

[1]Even though CUDA unified memory can use CPU memory, the communication cost can outweigh the parallelization payoff.

5.3.1 catDog and Analysis

The core operation in CNNs is *convolution* which constitutes the majority of the training time (approximately 60% of the compute time for ILSVRC winners [145]. It also typically consumes more memory than other network layers [137, 145]. cuDNN supports multiple algorithms for each type of convolution. For example, for forward convolution, it supports `GEMM`, `IMPLICIT_GEMM`, `IMPLICIT_PRECOMP_GEMM`, `WINOGRAD`, `WINOGRAD_NONFUSED`, `DIRECT`, `FFT`, and `FFT_TILING`. Depending on the convolution parameters (input, filter, data layout, etc.), each of the above algorithms has a different execution time, resource utilization and workspace memory. Convolutions in cuDNN use device global memory for storing input, output, filter, and intermediate results (or workspace). The input, output, and filter sizes for convolutions are fixed during model construction, so DL frameworks can only tune workspace memory.

Our experiments on numerous convolutions (from popular networks) reveal that it is not feasible to run two or more cuDNN convolutions concurrently. Using the Nvidia profiler, we observe that cuDNN kernels exhaust one or more SM resources (including registers and shared memory) and do not allow the GPU scheduler to execute blocks from another kernel on the same SM. Since a convolution typically has enough blocks to occupy all available SMs, execution of a second convolution is postponed to after the first convolution is completed resulting in a sequential execution of the two operations. Even though the profiler reports high occupancy for convolutions, for combinations of inputs and convolution algorithms, the computational efficiency and DRAM utilization are not high enough (e.g. 50%) [103, 107, 108, 104].

In addition, current DL frameworks either stick to certain algorithms for convolutions (e.g. MXNET) or pick the fastest algorithm (e.g. TensorFlow). For example, in the first iteration, TensorFlow tests all algorithms for each convolution and chooses the fastest one for subse-

quent iterations. Even though this method is optimal to reduce the execution time of linear networks, it is not essentially the best option for the parallel execution of operations since the fastest algorithm could inadequately use SM resources and/or consume a large amount of workspace memory preventing concurrent kernel executions. We observe this exact behavior by profiling the resource utilization and workspace memory of convolutions in popular networks.

SM resources. Table 5.1 shows profiling data for two independent convolutions in the inception module of GoogleNet. According to the table, PRECOMP_GEMM algorithms exhaust SM registers (more than 90%) but poorly use shared memory while FFT_TILING algorithms have complementary static resource utilization, i.e. bottlenecked by SM shared memory but consume only 38% of registers. Further, these two algorithms exhibit different warp execution characteristics. FFT_TILING has less than 30% ALU utilization but significantly greater memory stalls compared to the GEMM algorithm with high ALU utilization (70%) and lower memory stalls. This indicates, the former algorithm is relatively bound by memory rather than compute resources as in the latter algorithm.

In the past few years, researchers have proposed inter-SM [63] and intra-SM [81, 153, 150] partitioning to improve resource utilization for concurrent kernel executions. Spatial multitasking [63], which group SMs among kernels has performance benefits when kernels with complementary characteristics are co-located. In intra-SM partitioning, resource utilization is further improved by letting blocks from different kernels share the same SM. For instance, functional units in an SM (ALUs, SFUs, etc.) that are idle when running a memory-intensive kernel can be utilized by the blocks of a compute-intensive kernel. Intra-SM partitioning can practically be achieved when one or more SM static resources such as registers and shared memory remain under-utilized by kernels [81, 153]

Thereby, for two convolutions in Table 5.1, if we choose PRECOMP_GEMM for the first convolution and FFT_TILING for the second (TensorFlow would pick PRECOMP_GEMM

for both) and employ SM partitioning [81, 153], the memory stalls of the second convolution can potentially be hidden by switching to compute-warps from the first convolution. This parallelization can improve resource utilization and reduce latency compared to serial execution. We discover 27 similar cases in this network and more instances in other popular nonlinear CNNs such as ResNet.

Device Memory. Table 5.2 shows the execution time and workspace memory for a convolution operation in GoogleNet. Comparing the FFT algorithm (TensorFlow selection) with Winograd Nonfused, the former is only 21% faster but requires almost 1.5 GB (or 70%) of extra memory. Changing the convolution algorithm is the only way to configure workspace memory. Therefore, careful and profiling-based algorithm selection has the potential to mitigate concurrent kernel execution's limitations and improve the parallelism on a single GPU.

Algorithm (conv.)	GEMM	Implicit GEMM	Precomp GEMM	Winograd nonfused	FFT	FFT tiling
Memory	0	48 KB	4.8 GB	691 MB	2.2 GB	1.1 GB
Time	53 ms	59 ms	126 ms	46 ms	36 ms	48 ms

Table 5.2: Comparison of workspace memory and execution time for the convolution in the sixth layer of GoogleNet on K40 GPU using all algorithms in cuDNN. Direct and Winograd algorithms are not supported for this input.

We also go one step further to validate the benefit of using catDog. Unfortunately, we are unable to implement a tailor for DL operations on Nvidia GPUs because Nvidia does not provide a knob for partitioning compute resources. Also, neither cuDNN kernels nor Nvidia schedulers are open-source for instrumenting. AMD on the other hand recently enhanced their HIP stream APIs to preallocate CUs on GPU. With this new feature, we would be able to apply inter-SM (or inter-CU) partitioning and test catDog. AMD has also recently released ROCm which is an open-source platform for accelerated computing and includes equivalents for the majority of Nvidia ML libraries. For example, it includes hipDNN a

counterpart for cuDNN, though it is still premature.

By leveraging CU partitioning APIs and multi-stream support of ROCm in catDog, we are able to run multiple heavy kernels such as convolutions concurrently on an AMD GPU. We test catDog on the candidate convolutions from GoogleNet and verify the full concurrency with rocProf (ROCm profiler) on AMD MI50 GPU. Despite our success in parallel kernel execution (which so far has not been observed), the performance gain is insignificant. In all the cases, running two convolutions in parallel results in the same performance or marginally better runtime as compared to a sequential run. For example, running the two convolutions in Table 5.1 in parallel ($batchsize = 256$) takes 2.5 ms while sequential execution takes 2.6 ms. This tiny benefit is not surprising considering catDog still does not support intra-SM partitioning and more importantly, AMD does not support all the convolution algorithms for selecting the optimal algorithm (at the time of our measurements we were limited to GEMM and WINOGRAD).

M,K,N	α , β	Sequential exec. time		Parallel exec. time	
		4 GEMM	10 GEMM	4 GEMM	10 GEMM
$64, 1, 512$	$-1, 1$	1.0 ms	2.5 ms	1.0 ms	2.7 ms
$64, 512, 1$	$-1, 1$	1.5 ms	3.6 ms	1.0 ms	2.8 ms
$64, 512, 2048$	$-1, 1$	6.0 ms	15 ms	2.8 ms	6.9 ms
$1024, 64, 512$	$-1, 1$	3.5 ms	8.8 ms	1.7 ms	4.5 ms

Table 5.3: Parallel vs Sequential execution of GEMM operations in Attention module

Alternatively, we consider non-convolution operation, i.e., GEMM operations in attention units of BERT [83], a widely used model in NLP [78, 122]. Attention modules in NLP look at different positions in an input sentence and compute a representation of the sentence (i.e., dot product for encoding). Attention consists of a large number of light matrix-matrix multiplications. These multiplications are independent and potentially can run in parallel. So, we test catDog on multiple GEMMs in BERT's attention and observe significant performance gain from parallel execution of these kernels. Table 5.3 demonstrates the results

for 4 of these GEMMs. We observe parallel execution of these operations can result in more than 2× speedup. We also use catDog to find the optimal configuration in partitioning CUs among GEMM kernels. Figure 5.3 shows the performance of running 4 GEMMs (the last row from Table 5.3) in parallel as a function of the number of allocated CUs per kernel. This figure suggests assigning more than 28 CUs to each kernel which leads to high-performance training when parallelism is exploited. Less than 28 underutilizes the GPU's computing power. More precisely, the optimal configuration is 30 CUs/kernel, so our tailor references such profiling results and partitions CUs accordingly to maximize the performance gain.

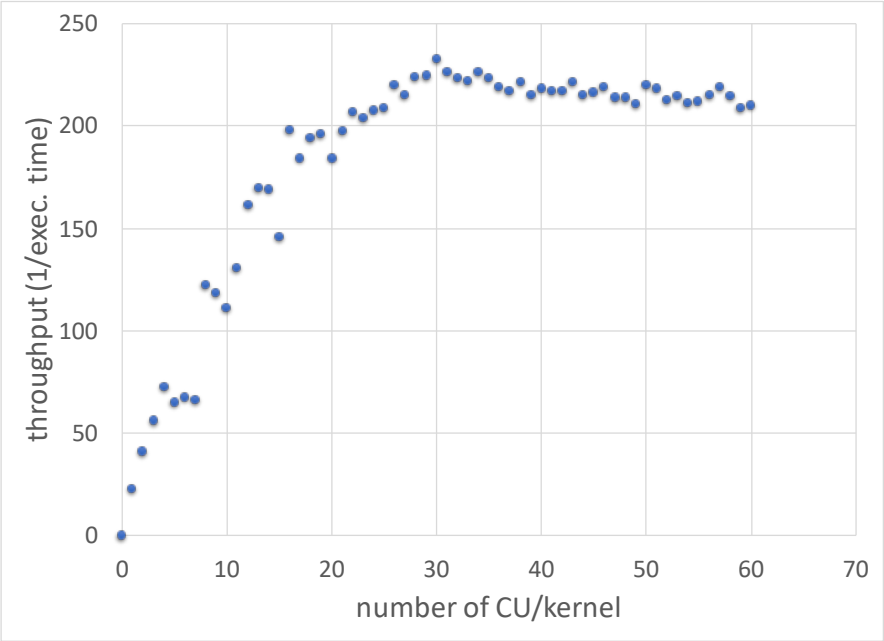


Figure 5.3: Performance of executing GEMM kernels (1024,64,512) concurrently based on share of CU.

5.4 Conclusions and Future Work

We conclude that partitioning GPU computing resources among concurrent convolutions depends on the workload (algorithm) which impacts both the execution time and workspace memory of kernels. There is also an inherent tradeoff between the execution time and workspace memory that leads to proper algorithm selection. Therefore, algorithm selection and resource allocation are mutually dependent. Even though we observe the potential for concurrent convolution execution, the proposed profiling-based algorithm selection should carefully evaluate multiple aspects for optimal parallelism. On the GEMM operations in attention units, however, we observe a necessity for parallelism. Our proposed method, catDog, can be extended to find concurrency opportunities more accurately and tailor operations to maximize performance. This enhancement highly depends on the existing software solutions. We hope in the future GPU manufacturers provide more powerful API and flexibility in their product, so we can implement catDog more efficiently.

Chapter 6

Concluding Remarks

6.1 Summary

We demonstrate four important applications of performance analysis in this dissertation. The primary contributions of these four applications are summarized below.

1. We conduct a what-if analysis of page load time in today's browser. The purpose of this study is to pinpoint the salient spots for performance optimization so that developers can focus their efforts there. We show the conventional PA of web browsers is not practical due to the scale and complexity of codebase, parallel and non-deterministic executing model of tasks, and the proliferation of task dependencies. For the first time, we adapt causal profiling to what-if analysis of such an application. To realize causal profiling for web browsers, we develop COZ+, a powerful profiler and performance analyzer that locates the impact of optimizing browser tasks on the page load time. We employ COZ+ on the 100 most visited websites and discover that optimizing only 40% of Scripting activities makes websites load 8.5% faster. Contrary to the community sentiment, optimizing HTML parsing and layout activities has a minor impact on PLT.

COZ+ also reveals that, while multi-tasking is prevalent in the rendering processes, there is not sufficient parallelism among tasks. Furthermore, we consider the impact of system architecture, caching, and network connection on the what-if analysis of PLT and discover consistency in critical stages in each configuration, and more importantly, the impact of optimization hardly alters under medium and high-speed internet connections, implying that browsers are currently bottlenecked by computation stages.

2. After observing promising results from applying causal profiling to complex applications, we extend this method for cross-platform PA and faster what-if analysis. Therefore, we propose virtual causal profiling, a scheme for translating the impact of code optimization across various devices. A key contribution of our research is to model causal profiling and prove the theory behind virtual causal profiling. We design and implement VCoz, a prototype for virtual causal profiling that throttles hardware components of a host system to simulate what-if analysis of programs on a target system. As a groundwork for our validation, we leverage the Coz profiler for android smartphones. We use VCoz and analyze parallel benchmarks consisting of CPU-intensive, memory-intensive, and I/O-intensive codes and compare the what-if graphs with the Coz profiler on a mobile device. The results validate the functionality of virtual causal profiling and show less than 16% variation with the ground truth.

3. Online advertising, which is prevalent in the web ecosystem and has caused performance concerns, is the third application of our performance analysis. We apply extensive performance characterization on third-party web ads to demystify their performance cost on the page loading process. We demonstrate that adblockers are ineffective at quantifying such performance costs, and subsequently develop adPerf, a rigorous performance analyzer for display ads. We employ adPerf in a large-scale study (on over 500 websites) and characterize the performance cost of web ads from three perspectives: 1) computation, 2) network and 3) delivery sources. We conclude that online advertising

has a 15% overhead on CPU usage and bandwidth usage. This is also applicable to mobile users but at a lower cost. Our conclusion is crucial for end-users who do not necessarily consent to receive such content. In our correlation assessments of performance and delivery sources, we measure that Google ads account for more than half of all client bandwidth from online advertising. Along with well-known and trusted online advertising players, there are a large number of untrusted or inadequately trusted domains that contribute to 37% of ads costs.

4. In our last application, we focus on enhancing the performance of deep learning model training. Modeling the performance of deep learning layers and demonstrating the need for inter-layer parallelization to reduce training time are two major contributions of our PA in this study. We introduce catDog that uses GPU profilers and characterizes the performance and resource utilization of each operation for every possible implementation/configuration. We test our performance analyzer (i.e., catDog) on popular non-linear CNNs such as GoogleNet and ResNet. Considering the resource constraints of the device, our results show that there is an optimal configuration for convolution layers of these networks that allows for parallel execution on a GPU. For instance, we show 27 cases in GoogleNet where selecting the optimal convolution algorithm can accommodate concurrent kernel execution. We also integrate inter-SM partitioning and multi-stream execution features in catDog for AMD GPUs and verify parallel execution of candidate convolutions in GoogleNet. Besides, we show that catDog can determine the optimal configuration and tailor GEMM operations in BERT attention modules so that parallel execution of these operations improves performance by up to 2×.

6.2 Future Directions

While hardware and software systems evolve, more advanced PA methods must be conceived and implemented to adapt to the complexity of future applications. Our work showcases multiple of these methodologies, but we believe that our performance analysis can be extended to a broader class of applications and future systems. Here we present two important directions for expanding our methods and frameworks.

6.2.1 Intelligent Causal Analysis

COZ+ directs developers where to spend their optimization efforts to maximize the performance gains. This tool is designed for large-scale and highly parallel applications. Along with that, VCoz informs developers how the desired optimization maps to other systems and configurations. In the future, the first step will be to expand the capabilities of the VCoz prototype to include more hardware components and combine it with the COZ+ profiler. The outcome will be a high-performance tool for causal analysis of a wide range of complex and multi-platform applications.

Although the final framework is expected to be a versatile tool for developers in identifying bottlenecks and optimization opportunities, it still cannot instruct them on how to optimize their code. For instance, if the most critical spot for optimization has a data-parallel structure, parallelizing the workload over multiple threads may be suggested, or if the computation is delayed due to data availability, splitting the workload and pipelining computation with data movement to hide latency may be suggested. Although designing such an intelligent causal analyzer is expected to be difficult, backend AI module can start with a set of predefined and widely used optimization techniques. It can communicate with the frontend profiler and exploit its performance data and assign a score to each technique.

The AI system may offer top N optimization techniques with the highest score. Alternatively, machine learning algorithms can be used in this direction to provide a higher level of confidence in the recommendation.

6.2.2 Causal Resource Profiler

In Chapter 5, we show how characterizing the performance of functions under different algorithms and system configurations reveals opportunities for parallelization. A brute-force scan of the configuration space is the method we currently use in catDog for such characterization. For example, to configure the optimal SM partitioning, catDog runs concurrent operations with every possible share of SMs and reports the execution time. Although this strategy generalizes to every application, as the applications become more complex, the search space expands dramatically, becoming a bottleneck for this approach.

One way to handle this problem is to design a performance analyzer that predicts the performance of a function for each configuration based on the resources the function uses in that configuration [126]. In other words, similar to the concept of causal profiling that predicts the application's performance from the changes in the runtime of a function (e.g., what if a function runs $X\%$ faster?), the new analyzer would predict the application's performance based on the changes in the resources used by a function (e.g., what if a function consumes $X\%$ less memory?). The causal resource profiler may throttle the resources used by the concurrent function to simulate the impact of allocating more/fewer resources to the candidate function. Implementing this causal resource profiler will undoubtedly necessitate extensive engineering, as we believe throttling resources at runtime without significantly impacting the application to be quite difficult.

Bibliography

[1] cublas. https://developer.nvidia.com/cublas.

[2] The Google Chrome web browser. http://www.google.com/chrome.

[3] Google developer: Analyzing critical rendering path performance. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/analyzing-crp.

[4] Internet explorer 9 network performance improvements. https://blogs.msdn.microsoft.com/ie/2011/03/17/internet-explorer-9-network-performance-improvements/.

[5] Cisco annual internet report (2018–2023) white paper, Feb 2017. https://www.statista.com/statistics/371894/average-speed-global-mobile-connection/.

[6] AD BLOCK DETECTION SCRIPT. https://iabtechlab.com/software/ad-block-detection-script/, 2020.

[7] Ad blocking user penetration rate in the United States. https://www.statista.com/statistics/804008/ad-blocking-reach-usage-us, 2020.

[8] AdBlock. https://getadblock.com/, 2020.

[9] Adblock Plus. https://adblockplus.org, 2020.

[10] adblockparser. https://github.com/scrapinghub/adblockparser, 2020.

[11] Alexa Top News Sites. https://www.alexa.com/topsites/category/News, June 2020.

[12] Alexa Top Sites. https://www.alexa.com/topsites/countries/US, June 2020.

[13] Android emulator, May 2020. https://developer.android.com/studio/run/emulator.

[14] Android NDK, May 2020. https://developer.android.com/ndk.

[15] Appetize, May 2020. https://appetize.io/.

[16] Average global mobile network connection speeds from 2016 to 2021, May 2020. https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[17] Chrome DevTools Protocol. https://chromedevtools.github.io/devtools-protocol, August 2020.

[18] Chrome Devtools Timeline. https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/timeline-tool, 2020.

[19] Chrome's coming changes to video ad blocking could impact YouTube. https://martechtoday.com/chromes-coming-changes-to-video-ad-blocking-could-impact-youtube-238360, 2020.

[20] Coz+, May 2020. https://gitlab.com/coz-plus.

[21] CPU performance scaling, May 2020. https://www.kernel.org/doc/html/v4.14/admin-guide/pm/cpufreq.html.

[22] EasyList. https://easylist.to, August 2020.

[23] Ghostery. https://www.ghostery.com/, 2020.

[24] Global internet advertising revenue in 2015 and 2020. https://www.statista.com/statistics/237800/global-internet-advertising-revenue/, 2020.

[25] Google android bionic, May 2020. https://android.googlesource.com/platform/bionic.

[26] Handling Heavy Ad Interventions. https://developers.google.com/web/updates/2020/05/heavy-ad-interventions, 2020.

[27] Heavy Ads: (brief description of issue). https://bugs.chromium.org/p/chromium/issues/detail?id=1114329, 2020.

[28] Lighthouse. https://developers.google.com/web/tools/lighthouse, 2020.

[29] Linux hdparm, May 2020. http://man7.org/linux/man-pages/man8/hdparm.8.html.

[30] Linux traffic control, May 2020. http://man7.org/linux/man-pages/man8/tc.8.html.

[31] Loading Third-Party JavaScript. https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/loading-third-party-javascript/?utm_source=lighthouse&utm_medium=unknown, 2020.

[32] Number of active desktop adblock plugin users worldwide. https://www.statista.com/statistics/435252/adblock-users-worldwide/, 2020.

[33] Oprofile - a system profiler for linux, May 2020. https://oprofile.sourceforge.io/.

[34] Popeteer. https://developers.google.com/web/tools/puppeteer/get-started, June 2020.

[35] simpleperf, May 2020. https://developer.android.com/ndk/guides/simpleperf.

[36] Stream benchmark, May 2020. http://www.cs.virginia.edu/stream/ref.html.

[37] uBlock. https://ublock.org/, 2020.

[38] VirusTotal. https://www.virustotal.com, June 2020.

[39] Website Safety, Security Check Web Of Trust. https://www.mywot.com/, 2020.

[40] Zbrowse. https://github.com/zmap/zbrowse, August 2020.

[41] Pagespeed insights, Accessed: 2018-07-20. https://developers.google.com/speed/pagespeed/insights/.

[42] Chrome devtool, Accessed: 2019-01-25. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.

[43] Chrome: Process model, Accessed: 2019-01-25. https://www.chromium.org/developers/design-documents/process-models.

[44] Gecko. https://developer.mozilla.org/en-US/docs/Mozilla/Gecko, Accessed: 2019-01-25.

[45] Gecko profiler, Accessed: 2019-01-25. https://developer.mozilla.org/en-US/docs/Mozilla/Performance/Profiling_with_the_Built-in_Profiler.

[46] Linux perf, Accessed: 2019-01-25. http://man7.org/linux/man-pages/man1/perf.1.html.

[47] Microsoft edge. https://docs.microsoft.com/en-us/microsoft-edge/, Accessed: 2019-01-25.

[48] The rendering critical path, Accessed: 2019-01-25. https://www.chromium.org/developers/the-rendering-critical-path.

[49] Spdy, Accessed: 2019-01-25. http://dev.chromium.org/spdy.

[50] Spidermonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey, Accessed: 2019-01-25.

[51] The trace event profiling tool. https://www.chromium.org/developers/how-tos/trace-event-profiling-tool, Accessed: 2019-01-25.

[52] Trident(mshtml reference), Accessed: 2019-01-25. https://docs.microsoft.com/en-us/previous-versions/windows/internet-explorer/ie-developer/platform-apis/aa741317(v=vs.85).

[53] V8. https://developers.google.com/v8/, Accessed: 2019-01-25.

[54] The WebKit open source project, Accessed: 2019-01-25. http://www.webkit.org.

[55] The most popular browsers, December 2018. https://www.w3schools.com/browsers/.

[56] Statcounter global browser stats. http://gs.statcounter.com/browser-market-sharemonthly-201802-201802-bar, December 2018.

[57] Cuda c++ programming guide, May 2021. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.htmll.

[58] Hip programming guide, May 2021. https://rocmdocs.amd.com/en/latest/Programming_Guides/HIP-GUIDE.html.

[59] Yslow, Released: 2012. https://yslow.org.

[60] Blink. https://www.chromium.org/blink, Released: 2013.

[61] Coz, Released: 2015. https://github.com/plasma-umass/coz.

[62] O. Abdel-Hamid, A.-r. Mohamed, H. Jiang, L. Deng, G. Penn, and D. Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Transactions on audio, speech, and language processing*, 22(10):1533–1545, 2014.

[63] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The case for gpgpu spatial multitasking. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–12. IEEE, 2012.

[64] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin. Engineering the servo web browser engine using rust. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 81–89. ACM, 2016.

[65] G. Anthes. Data brokers are watching you. *Commun. ACM*, 58(1):28–30, 2014.

[66] C.-A. Avram, K. Salem, and B. Wong. Latency amplification: Characterizing the impact of web page content on load times. In *Reliable Distributed Systems Workshops (SRDSW), 2014 IEEE 33rd International Symposium on*, pages 20–25. IEEE, 2014.

[67] C. Badea, M. R. Haghighat, A. Nicolau, and A. V. Veidenbaum. Towards Parallelizing the Layout Engine of Firefox. In *Proc. of the 2nd USENIX Conf. on Hot topics in parallelism*, pages 1–1. USENIX Assoc., 2010.

[68] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium*, pages 481–496, 2016.

[69] F. Bellard. Qemu, a fast and portable dynamic translator.

[70] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[71] H. Binsalleeh. *Analysis of Malware and Domain Name System Traffic*. PhD thesis, Concordia University, 2014.

[72] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328. ACM, 2011.

[73] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 313–328, 2011.

[74] C. Cascaval, S. Fowler, P. Montesinos-Ortego, W. Piekarski, M. Reshadi, B. Robatmili, M. Weber, and V. Bhavsar. 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, pages 271–280, New York, NY, USA, 2013. ACM.

[75] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[76] P. H. Chia and S. J. Knapskog. Re-evaluating the wisdom of crowds in assessing web security. In *International Conference on Financial Cryptography and Data Security*, pages 299–314. Springer, 2011.

[77] N. Chidambaram Nachiappan, P. Yedlapalli, N. Soundararajan, M. T. Kandemir, A. Sivasubramaniam, and C. R. Das. Gemdroid: a framework to evaluate mobile platforms. *ACM SIGMETRICS Performance Evaluation Review*, 42(1):355–366, 2014.

[78] K. Clark, U. Khandelwal, O. Levy, and C. D. Manning. What does bert look at? an analysis of bert's attention. *arXiv preprint arXiv:1906.04341*, 2019.

[79] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167, 2008.

[80] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197. ACM, 2015.

[81] H. Dai, Z. Lin, C. Li, C. Zhao, F. Wang, N. Zheng, and H. Zhou. Accelerate gpu concurrent kernel execution by mitigating memory pipeline stalls. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 208–220. IEEE, 2018.

[82] A. De Corniere and R. De Nijs. Online advertising and privacy. *The RAND Journal of Economics*, 47(1):48–72, 2016.

[83] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[84] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1388–1401. ACM, 2016.

[85] D. S. Evans. The online advertising industry: Economics, evolution, and privacy. *Journal of economic perspectives*, 23(3):37–60, 2009.

[86] S. Eyerman, K. Du Bois, and L. Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 145–155. IEEE, 2012.

[87] F. Farahmand, J. Wan, J. Kistner-Morris, and N. Gabor. Photoresponse of a dirac-weyl interface composed of graphene and wte 2. *Bulletin of the American Physical Society*, 2021.

[88] C. Fernando, D. Banarse, C. Blundell, Y. Zwols, D. Ha, A. A. Rusu, A. Pritzel, and D. Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *arXiv preprint arXiv:1701.08734*, 2017.

[89] D. Fisher, B. Wilson, B. Goodger, and A. Weber. Multi-process browser architecture, Oct. 16 2012. US Patent 8,291,078.

[90] B. Franke. Fast cycle-approximate instruction set simulation. In *Proceedings of the 11th international workshop on Software & compilers for embedded systems*, pages 69–78, 2008.

[91] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[92] K. Garimella, O. Kostakis, and M. Mathioudakis. Ad-blocking: A study on performance, privacy and counter-measures. In *Proceedings of the 2017 ACM on Web Science Conference*, pages 259–262. ACM, 2017.

[93] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[94] M. Ikram and M. A. Kaafar. A first look at mobile ad-blocking apps. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2017.

[95] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi. The chain of implicit trust: An analysis of the web third-party resources loading. In *The World Wide Web Conference*, pages 2851–2857. ACM, 2019.

[96] U. Iqbal, Z. Shafiq, and Z. Qian. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference*, pages 171–183. ACM, 2017.

[97] U. Iqbal, Z. Shafiq, P. Snyder, S. Zhu, Z. Qian, and B. Livshits. Adgraph: A machine learning approach to automatic and effective adblocking. *arXiv preprint arXiv:1805.09155*, 2018.

[98] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale. Evaluating hpc networks via simulation of parallel workloads. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165. IEEE, 2016.

[99] C. G. Jones, R. Liu, L. Meyerovich, K. Asanovic, and R. Bodik. Parallelizing the web browser. In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[100] M. Ju, H. Kim, and S. Kim. Mofysim: A mobile full-system simulation framework for energy consumption and performance analysis. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 245–254. IEEE, 2016.

[101] S. L. G. P. B. Kessler and M. K. McKusick. gprof: a call graph execution profiler. In *Proceedings of the Symposium on Compiler Construction*. Citeseer, 1982.

[102] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[103] A. Lavin. maxDNN: An efficient convolution kernel for deep learning with maxwell GPUs. *arXiv preprint arXiv:1501.06633*, 2015.

[104] A. Lavin and S. Gray. Fast algorithms for convolutional neural networks. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[105] Y. LeCun, K. Kavukcuoglu, C. Farabet, et al. Convolutional networks and applications in vision. In *ISCAS*, volume 2010, pages 253–256, 2010.

[106] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium*, 2016.

[107] J. Lew, D. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. Aamodt. Analyzing machine learning workloads using a detailed GPU simulator. *arXiv preprint arXiv:1811.08933*, 2018.

[108] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou. Optimizing memory efficiency for deep convolutional neural networks on GPUs. In *Proc. of the Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, pages 633–644. IEEE, 2016.

[109] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 674–686. ACM, 2012.

[110] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. G. Greenberg, and Y.-M. Wang. Webprophet: Automating performance prediction for web services. In *NSDI*, volume 10, pages 143–158, 2010.

[111] L. A. Meyerovich and R. Bodik. Fast and parallel webpage layout. In *Proceedings of the 19th international conference on World wide web*, pages 711–720. ACM, 2010.

[112] J. Mickens. Silo: Exploiting javascript and dom storage for faster page loads. In *WebApps*, 2010.

[113] B. Mostafazadeh Davani, F. Marti, B. Pourghassemi, F. Liu, and A. Chandramowlishwaran. Unsteady navier-stokes computations on gpu architectures. In *23rd AIAA Computational Fluid Dynamics Conference*, page 4508, 2017.

[114] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns. Enabling parallel simulation of large-scale hpc network systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):87–100, 2016.

[115] M. H. Mughees, Z. Qian, and Z. Shafiq. Detecting anti ad-blockers in the wild. *Proceedings on Privacy Enhancing Technologies*, 2017(3):130–146, 2017.

[116] J. Nejati and A. Balasubramanian. An In-depth study of Mobile Browser Performance. In *Proc. of the 25th Intl. Conf. on WWW*, pages 1305–1315. Intl. WWW Conf. Steering Committee, 2016.

[117] R. Netravali, A. Goyal, J. Mickens, and H. Balakrishnan. Polaris: Faster page loads using fine-grained dependency tracking. In *NSDI*, pages 123–136, 2016.

[118] R. Nithyanand, S. Khattak, M. Javed, N. Vallina-Rodriguez, M. Falahrastegar, J. E. Powles, E. De Cristofaro, H. Haddadi, and S. J. Murdoch. Adblocking and counter blocking: A slice of the arms race. In *6th USENIX Workshop on Free and Open Communications on the Internet (FOCI 16)*, 2016.

[119] O. Obiols-Sales, A. Vishnu, N. Malaya, and A. Chandramowliswharan. Cfdnet: A deep learning-based accelerator for fluid simulations. In *Proceedings of the 34th ACM International Conference on Supercomputing*, pages 1–12, 2020.

[120] D. Pelenis, D. Barauskas, G. Vanagas, M. Dzikaras, and D. Viržonis. Cmut-based biosensor with convolutional neural network signal processing. *Ultrasonics*, 99:105956, 2019.

[121] P. Pezeshkpour, L. Chen, and S. Singh. Embedding multimodal relational data for knowledge base completion. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2018.

[122] P. Pezeshkpour, S. Jain, B. C. Wallace, and S. Singh. An empirical comparison of instance attribution methods for nlp. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 967–975, 2021.

[123] B. Pourghassemi, A. Amiri Sani, and A. Chandramowlishwaran. What-if analysis of page load time in web browsers using causal profiling. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–23, 2019.

[124] B. Pourghassemi, J. Bonecutter, Z. Li, and A. Chandramowlishwaran. adperf: Characterizing the performance of third-party ads. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(1):1–26, 2021.

[125] B. Pourghassemi and A. Chandramowlishwaran. cudacr: An in-kernel application-level checkpoint/restart scheme for cuda-enabled gpus. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 725–732. IEEE, 2017.

[126] B. Pourghassemi, J. H. Lee, and Y. S. KI. Platform for concurrent execution of gpu operations, July 23 2020. US Patent App. 16/442,440.

[127] B. Pourghassemi, A. A. Sani, and A. Chandramowlishwaran. Only relative speed matters: Virtual causal profiling. *ACM SIGMETRICS Performance Evaluation Review*, 2021.

[128] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, pages 93–106. ACM, 2015.

[129] S. Radhakrishnan, Y. Cheng, J. Chu, A. Jain, and B. Raghavan. Tcp fast open. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies*, page 21. ACM, 2011.

[130] M. Z. Rafique, T. Van Goethem, W. Joosen, C. Huygens, and N. Nikiforakis. It's free for a reason: Exploring the ecosystem of free live streaming services. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS 2016)*, pages 1–15. Internet Society, 2016.

[131] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. Ieee, 2007.

[132] D. Sahasrabudhe, R. Zambre, A. Chandramowlishwaran, and M. Berzins. Optimizing the hypre solver for manycore and gpu architectures. *Journal of Computational Science*, 49:101279, 2021.

[133] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111. IEEE, 2016.

[134] R. Simons and A. Pras. The hidden energy cost of web advertising. In *Proceedings of the 12th Twente Student Conference on Information Technology*, pages 1–8, 2010.

[135] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[136] P. Snyder, A. Vastel, and B. Livshits. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(2):1–24, 2020.

[137] D. Strigl, K. Kofler, and S. Podlipnig. Performance and scalability of GPU-based convolutional neural networks. In *18th Euromicro International Conf. on Parallel, Distributed and Network-Based Processing*, pages 317–324. IEEE, 2010.

[138] D. Sunwoo, W. Wang, M. Ghosh, C. Sudanthi, G. Blake, C. D. Emmons, and N. C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. In *2013 IEEE International Symposium on Workload Characterization*, pages 113–122. IEEE, 2013.

[139] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[140] L. Tang, Y. Wang, T. L. Willke, and K. Li. Scheduling computation graphs of deep learning models on manycore CPUs. *arXiv preprint arXiv:1807.09667*, 2018.

[141] A. Tazarv, S. Labbaf, S. M. Reich, N. Dutt, A. M. Rahmani, and M. Levorato. Personalized stress monitoring using wearable sensors in everyday settings. *arXiv preprint arXiv:2108.00144*, 2021.

[142] A. Tazarv and M. Levorato. A deep learning approach to predict blood pressure from ppg signals. *arXiv preprint arXiv:2108.00099*, 2021.

[143] C.-H. Tu, H.-H. Hsu, J.-H. Chen, C.-H. Chen, and S.-H. Hung. Performance and power profiling for emulated android systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 19(2):1–25, 2014.

[144] H. Wang, R. Planas, A. Chandramowlishwaran, and R. Bostanabad. Train once and use forever: Solving boundary value problems in unseen domains with pre-trained deep learning models. *arXiv preprint arXiv:2104.10873*, 2021.

[145] L. Wang, J. Ye, Y. Zhao, W. Wu, A. Li, S. L. Song, Z. Xu, and T. Kraska. Superneurons: Dynamic GPU memory management for training deep neural networks. In *Proc. of the ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 41–53. ACM, 2018.

[146] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with wprof. In *NSDI*, pages 473–485, 2013.

[147] X. S. Wang, A. Krishnamurthy, and D. Wetherall. Speeding up web page loads with shandian. In *NSDI*, pages 109–122, 2016.

[148] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. Why Are Web Browsers Slow on Smartphones? In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, New York, NY, USA, 2011. ACM.

[149] Z. Wang, F. X. Lin, L. Zhong, and M. Chishtie. How far can client-only solutions go for mobile browser speed? In *Proceedings of the 21st international conference on World Wide Web*, pages 31–40. ACM, 2012.

[150] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 358–369. IEEE, 2016.

[151] C. E. Wills and D. C. Uzunoglu. What ad blockers are (and are not) doing. In *2016 Fourth IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 72–77. IEEE, 2016.

[152] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[153] Q. Xu, H. Jeon, K. Kim, W. W. Ro, and M. Annavaram. Warped-slicer: efficient intra-sm slicing through dynamic resource partitioning for gpu multiprogramming. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 230–242. IEEE, 2016.

[154] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[155] A. Yoga and S. Nagarakatte. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 485–501, 2019.

[156] H. W. Yu, R. Chen, H. Wang, Z. Yuan, Y. Zhao, Y. An, Y. Xu, and L. Zhu. Gpu accelerated lattice boltzmann simulation for rotational turbulence. *Computers & Mathematics with Applications*, 67(2):445–451, 2014.

[157] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol. Tracking the trackers. In *Proceedings of the 25th International Conference on World Wide Web*, pages 121–132, 2016.

[158] R. Zambre, L. Bergstrom, L. A. Beni, and A. Chandramowlishwaran. Parallel performance-energy predictive modeling of browsers: Case study of servo. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 22–31. IEEE, 2016.

[159] K. Zhang, L. Wang, A. Pan, and B. B. Zhu. Smart caching for web browsers. In *Proceedings of the 19th international conference on World wide web*, pages 491–500. ACM, 2010.

[160] M. Zhang, Y.-M. Wang, A. Greenberg, and L. Zhichun. Web page load time prediction and simulation, Dec. 18 2012. US Patent 8,335,838.

[161] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen. HPar: A practical parallel parser for HTML–taming HTML complexities for parallel parsing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 10(4):44, 2013.

[162] S. Zhu, U. Iqbal, Z. Wang, Z. Qian, Z. Shafiq, and W. Chen. Shadowblock: A lightweight and stealthy adblocking browser. In *The World Wide Web Conference*, pages 2483–2493. ACM, 2019.