Public Review for

Mobile Web Browsing Under Memory Pressure

Ihsan Ayyub Qazi, Zafar Ayyub Qazi, Theophilus A. Benson, Ehsan Latif, Abdul Manan, Ghulam Murtaza, Muhammad Abrar Tariq

Smartphones are today the primary web browsing and interaction point for the majority of the Internet users. However, the increasing complexity of webpages, and the large number of external elements and objects often lead to major bottlenecks in mobile browsing, and for commercial reasons many webpages do not offer a lean mobile version.

In this paper, the authors investigate the impact of memory usage on mobile devices in the context of web browsing. Through a number of studies the authors characterise the effects of items on the page, images, other apps, and device specs on page load time. This is a rapidly changing space. The work here presents a study using landing page loading time and memory requirements for a number of Android-based smartphones using Chrome, Firefox, Microsoft Edge and Brave. In addition , the paper presents an extensive set of results on the effect of tabs, scrolling, the number of images, and the umber of requests made for different objects.

The findings in the paper have a number of implications for browser vendors and webpage designers when choosing various content types and formats. The reviewers found the study and its results interesting, and recommended a number of further studies which the authors addressed to prepare the final version of the paper. I hope the findings will provide valuable insights for the mobile web practitioners, alongside enabling further research and measurement studies in this space.

Public review written by
Hamed Haddadi
Imperial College London, UK

Mobile Web Browsing Under Memory Pressure

Ihsan Ayyub Qazi, Zafar Ayyub Qazi, Theophilus A. Benson*, Ghulam Murtaza, Ehsan Latif, Abdul Manan, Abrar Tariq
LUMS, *Brown University
{ihsan.qazi,zafar.qazi,ehsan.latif,20100198,20100262,19100088}@lums.edu.pk,theophilus_benson@brown.edu

ABSTRACT

Mobile devices have become the primary mode of Internet access. Yet, differences in mobile hardware resources, such as device memory, coupled with the rising complexity of Web pages can lead to widely different quality of experience for users. In this work, we analyze how device memory usage affects Web browsing performance. We quantify the memory footprint of popular Web pages over different mobile devices, mobile browsers, and Android versions, analyze the induced memory distribution across different browser components (e.g., JavaScript engine and compositor), investigate how performance gets impacted under memory pressure and propose optimizations to reduce the memory footprint of Web browsing. We show that these optimizations can improve performance and reduce chances of browser crashes in low memory scenarios.

CCS CONCEPTS

• **Information systems** → **Browsers**; • **Hardware**; • **Computing methodologies**;

KEYWORDS

Mobile, Web Browsing, QoE, Device Memory

1 INTRODUCTION

Mobile devices have become the dominant mode for Internet access [14]. Since October 2016, more websites have been loaded on mobile devices than on desktop computers [37]. However, we see large differences in mobile hardware resources (e.g., memory size, CPU speed, and the number of CPU cores). For example, in the year 2018, approximately 300 million Android devices shipped globally had 1 GB or less memory, whereas 400 million Android devices had at least 4 GB of memory [9]. Such heterogeneity in device resources makes it challenging for website operators to deliver a uniform experience to users.

At the same time, mobile Web pages are becoming increasingly complex, which is placing an increasing burden on device resources such as memory. For example, in the last seven years, the median Web page size has increased from 302 KB to 1748 KB, resulting in a median mobile Web page taking 18.7 s to load, which is 300% longer than the median desktop page [13].

Given these trends, it is becoming increasingly likely for devices to operate in regimes where the available memory becomes too low for a device to operate efficiently. Lack of available memory can potentially degrade the mobile quality of experience (QoE), prevent feature-rich applications to run, or lead to application crashes [3]. This situation can be particularly problematic for entry-level devices that have small RAM sizes, to begin with. Poor application performance in such regimes can frustrate users and reduce incentives for Original Equipment Manufacturers (OEMs) to manufacture entry-level devices, thereby potentially excluding many users from the smartphone ecosystem.

Several ongoing and recent efforts aim to improve mobile QoE [19, 32, 35, 37, 39, 40, 45] including efforts that solely target entry-level devices [4, 10, 42]. These solutions include the development of tools for constructing simplified mobile Web pages (e.g., AMP [1, 37]), delivering light pages to users either through a proxy-based transcoding service such as Google's Web Light [29] or through server-side customization (e.g., Free Basics [10]), and specialized operating systems for entry-level devices (e.g., Android Go [4]). While there has been anecdotal evidence [12, 17] that application performance is affected when a device is running low on memory, we are not aware of prior, independent efforts that aim to *quantify* the impact of low memory on the QoE of mobile Web browsing.

In this work, we analyze *how* and *when* device memory usage (also known as memory pressure) affects Web browsing performance. To this end, we (i) quantify the memory footprint of Alexa top 100 Web pages across different use cases (e.g., multiple tabs and page scrolling), mobile browsers (e.g., Chrome, Firefox, Microsoft Edge and Brave), mobile devices (e.g., devices with different RAM sizes and the number of CPU cores as shown in Table 3), and Android versions (e.g., Android 6, 7, and 8, and Android Go [4]), (ii) analyze their memory distribution across different components (e.g., JavaScript engine and compositor), (iii) investigate how these performance gets impacted under memory pressure and (iv) propose optimizations to reduce the memory footprint of Web pages. To ensure experimental precision, we conduct

Key Measurement Insights	Optimizations
(1) Mobile Chrome resulted in the least average memory footprint for loaded Web pages (Alexa top 100) compared to Brave, Microsoft Edge, and Firefox browsers. (2) Median memory footprint of Alexa top 100 pages on Chrome was 45x *larger* than the Web page size (3) Chrome Compositor and JS engine take up most memory across majority of Web pages (4) With newer Android versions (while keeping the same Chrome version), the average memory footprint of Alexa top 100 pages increased by 7.6 MB and 9.8 MB, with Android 7 and Android 8, respectively over Android 6. (5) Under critical memory pressure, the average PLT increases by 28.2% (5% of the pages experience more than 100% increase in PLT). We find that this trend holds across different mobile devices and Android versions. (6) 6.3% of the Web pages crashed under critical memory pressure (7) Under critical memory pressure, having more CPU cores can substantially improve PLTs due to reduced interference with system daemons	(1) Using browser native JS reduced Java heap size by 1.6x-12.3x and 1.1x-9.9x and the scripting time by 3x-105x and 4x-152x over using jQuery and React, respectively (2) Debloating jQuery reduced reduced Java heap size by 1 MB and reduced the loading time by 100 ms (3) On a synthetic page with 16 images, replacing PNG and GIF images with either JPEG or WebP versions reduced the memory footprint by 12%-45% and 12%-35%, respectively

Table 1: Key insights and optimization based on our analysis.

controlled experiments (§3), and use a combination of tools (e.g., Google's Telemetry [23][1] and Web Page Replay [30]) to isolate and attribute resource usage to individual browser components.

Table 1 provides a summary of our key insights and optimizations. We find that for Alexa top 100 Web pages, the memory footprint of the median page was at least 45x *larger* than the page size; with the compositor and the JavaScript (JS) engine contributing the most to memory usage across the majority of websites. We find that mobile Chrome induces the least average memory overhead for Alexa top 100 pages compared to Firefox, Microsoft Edge, and Brave. We also find that Web page loads slow down significantly when a device is operating under low memory (e.g., we observe an increase of 28% in the average page load time) with complex Web pages frequently crashing (6.3% of Web pages crashed in our experiments). While the former occurs due to reclaiming of cached pages by the kernel swap daemon (*kswapd*) when the amount of *free* memory falls below a certain threshold, the latter happens when the low memory killer daemon (*lmkd*) kills a process due to low memory. This slowing of page loads under high memory pressure holds across different mobile devices and Android versions.

We present optimizations to improve Web browsing performance and reduce the chances of Web page crashes under low memory regimes. For example, we find that debloating JS libraries (e.g., by removing unused functions), using browser native JS, and employing memory-efficient image formats can significantly reduce the memory footprint of a Web page. For example, we find that jQuery (55%) and React (8%) are the two most popular JS libraries used in Alexa top 880 Web pages. We show that for common DOM operations, replacing JS code that relies on external JS libraries with browser native JS code can reduce the JS heap size by 1.6x-12.3x and 1.1x-9.9x and the scripting time by 3x-105x and 4x-152x compared to using jQuery and React (two most popular JS libraries in our dataset), respectively.

Our work has implications for different stakeholders in the smartphone ecosystem. For website developers, our work provides insights about how websites can be designed to run better on low memory devices. In particular, we identify which application components contribute the most to memory, how the device responds in low memory scenarios, and which features can be adapted to reduce performance degradation and prevent crashes.

For OEMs, our work provides insights about hardware features that impact popular applications (e.g., *how beneficial is it to have more CPU cores when a device is running low on memory?*). For example, we find that under high memory pressure regimes using 4 cores compared to 2 cores improved the average PLT by 51%-69% across different Web pages. This happens because high priority system daemons (e.g., *kswapd* and *lmkd*) steal CPU cycles from the foreground browser application. This is unlike low-memory pressure scenarios where 1-2 cores are sufficient for achieving high performance. For users, our work can inform them about when Web browsing starts to perform badly (and how users' actions impact application behavior). Finally, for system developers our work sheds light on system-level mechanisms that can help improve performance (e.g., *kswapd* scheduling).

The remainder of this paper is organized as follows: we first motivate this study through a series of experiments (§2) and then describe our experimental methodology in detail (§3). We measure the memory footprint of Alexa top 100 pages over several mobile phones, mobile browsers, and Android versions (§4). We then analyze and quantify the impact of memory pressure on mobile QoE and browser crashes (§5). We present our optimizations to reduce the memory footprint of Web browsing under high memory pressure (§6). We then discuss the implications of our work for different stakeholders in the smartphone ecosystem (§7), discuss related work (§8), and offer concluding remarks (§9).

[1] Telemetry is the performance testing framework used by Chrome.

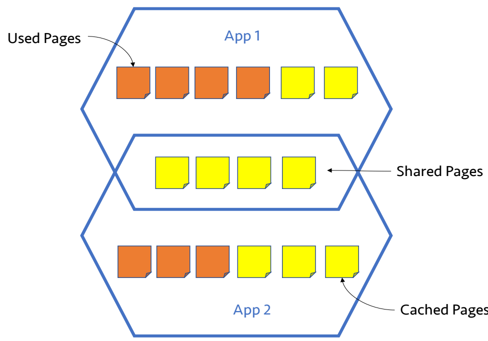


Figure 1: An example of two application processes with used pages, cached pages, and shared pages. The PSS of both processes is 8 (i.e., 6 + 4/2).

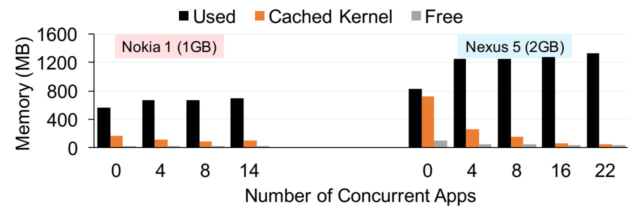


Figure 2: State of device memory when different number of mobile apps are concurrently run on 1 GB and 2 GB memory devices. We pick the most popular mobile apps on Google Play Store for this experiment.

Device	Open Apps	Avg. PLT (secs)	Increase in PLT (%)
Nokia 1	1, 14	6.1, 7.5	17
Nexus 5	1, 22	4.9, 8.8	42.1

Table 2: Average PLT for five randomly selected Web pages from Alexa top 100 when 1 and 14 apps are open in the background under Nokia 1 and 1 and 22 apps with Nexus 5.

We have shared all the code and data needed to replicate the experimental results in this paper in a public repository. The details are available in the appendix section.

2 MOTIVATION

In this section, we highlight how applications can be impacted by low device memory through a series of experiments. The subsequent sections describe in detail our experimental methodology (§3) and the complete set of experimental results across different configurations, OS versions, and phone models (§4 and §5). We begin this section by briefly describing how memory is organized on mobile devices.

Physical memory. The memory of a device is typically divided into fixed size pages (e.g., 4 kB pages) that can be used by different processes [21]. There are three types of pages: (1) *used pages*, which are being actively used by processes, (2) *cached pages*, which are being used by processes but the data they contain is also backed up in the device storage, thus these pages can be reclaimed if needed. These pages are further divided into cached pages used by mobile apps (whose size is denoted by *cached PSS*) and the kernel (*cached kernel*) and (c) *free pages*, which are pages that are not being used for anything. Note that system memory is used by both the CPU and the device GPU (if any).

We measure the memory used by an application by its Proportional Set Size (PSS), which is the portion of memory occupied by a process and is composed of the private memory of that process plus the proportion of shared memory with one or more other processes. We use PSS because it avoids over-counting or under-counting the memory impact of shared pages as shown in Figure 1.

Memory size limits the number of concurrent apps. To analyze how free memory changes with the number of open apps, we consider a 1 GB device (Nokia 1) and a 2 GB device (Nexus 5)[2]. We pick the most popular apps from Google Play Store, which include Gmail, Chrome, WhatsApp, YouTube, Twitter, and Hangout among others [11]. We find that the 1 GB device can support at most 16 apps concurrently (with one app in the foreground and the rest in the background) and immediately kills app(s) if any more are opened. On the 2 GB device, we could run at most 26 apps concurrently. These numbers can vary depending on the memory footprint of individual opened apps. Observe that in both phones, the number of free and/or cached pages decreases as new apps are opened as shown in Figure 2[3].

Memory usage affects page load times. To illustrate how the number of open apps in the background affects page load performance, we randomly pick five Web pages from Alexa top 100 and load them in the browser. We compare the average PLT of the pages on Nokia 1 and Nexus 5 phones. With Nokia 1, we run these experiments with 1 and 14 background apps, and with Nexus 5, we consider 1 and 22 apps. We pick 14 and 22 apps because the memory pressure becomes high/critical when these many apps are opened on the respective phones. Observe that the average PLT increases by 17% when using Nokia 1 and 42.1% on Nexus 5, as shown in Table 2.

[2]The Nokia 1 phone runs Android 8.0.1 (Go edition) whereas the Nexus 5 phone runs Android 6.0.1, the highest Android version supported by the phone. We use the default device/OS configuration for these experiments.
[3]For clarity, we do not show the *cached PSS* in the figure. Unlike *cached kernel* pages, these *cached PSS* can only be reclaimed if the app is killed by the system, reduces its memory footprint, or is explicitly closed.

Device Name	CPU Cores	Clock Min-Max (GHz)	RAM (GB)	GPU	OS Version	Release Cost	Release Date
Nexus 6P	8	1.55-1.95	3	Adreno 430 (650MHz)	6.0.0 - 8.1.0	$499	Sept, 2015
Nexus 6	4	2.7	3	Adreno 420 (600MHz)	5.0.1 - 7.1.1	$649	Oct, 2014
Nexus 5X	6	1.44-1.8	2	Adreno 418 (600MHz)	6.0.0 - 8.1.0	$379	Sept, 2015
Nexus 5	4	2.26	2	Adreno 330 (450MHz)	4.4 - 6.0.1	$349	Oct, 2013
Nokia 1	4	1.1	1	Mali-T720MP1 (600MHz)	8.1.0 Go	$99	Feb, 2018

Table 3: Mobile devices used in our experiment and their corresponding specifications.

3 METHODOLOGY

In this section, we describe the experimental methodology we devised to understand the impact of memory on the performance of Web browsing. One key challenge was to ensure repeatability in our experiments. Below we describe our testbed configuration, experimental design, and the choice of performance metrics.

We measure the memory footprint of the top 100 Web pages from Alexa over different mobile browsers (e.g., Chrome, Firefox, Microsoft Edge, and Brave). We then conduct a deeper analysis of memory usage in Chrome. We focus on the Chromium browser (an open-source version of Chrome) for two key reasons: (1) Chrome is the most popular mobile browser [25] and (2) many mobile browsers are based on Chromium including Brave [8], Microsoft Edge [18], Opera [26], Vivaldi [27], and Kiwi [16].

We obtain fine-grained memory measurements (e.g., the memory consumed by Java heap and the compositor) and page load times using telemetry, the performance testing framework used by Chrome [23]. To control network conditions and allow for repeatability in experiments, we record and replay Web pages using telemetry's Web Page Replay service. PLT is the time elapsed between when the URL is sent to the server and when the onload event is fired – which marks the point at which all resources that a page requires have been downloaded and processed – thus capturing both network loading and the device rendering time. Alternatives to PLT such as the above-the-fold time (AFT) and speed index [43] represent user-perceived page load times and measure the time it takes for the visible parts of a page to be displayed. However, they require screen/video recordings which can increase memory and CPU usage and thus confound the true memory consumption of mobile Web pages [22]. We host the pages on a desktop and set the download speed to 30 Mbps and the upload speed to 15 Mbps. We clear browser cache and cookies between any two page loads.

For each use case and workload, we repeat the experiment 10 times, unless stated otherwise. The metrics we measure include PSS, PLT and the fraction of Web pages that crash due to low memory. Unless noted otherwise, we present average values for different metrics. While we conduct our evaluation across several phones as shown in Table 3[4], we use Nexus 5 for most of our experiments, a device with a 2 GB memory. We choose Nexus 5 for the generality of our analysis as performance can be worse on devices with 1 GB memory, or less. Moreover, Nexus 5 was reported to be among the ten most popular Android smartphones in several countries in 2019 [24].

4 MEMORY USAGE OF WEB BROWSING

In this section, we analyze the memory footprint of Web browsing when loading Web pages with different complexities. We first measure the memory consumed by different mobile browsers (e.g., Chrome, Firefox, Microsoft Edge and Brave). We then conduct a deeper analysis of memory usage in Chrome. We analyze how memory usage gets distributed across different browser processes (e.g., Renderer and Browser processes), components (e.g., Java Heap and Compositor) and use cases (e.g., page scrolling and when using multiple tabs) and analyze factors that contribute to larger memory consumption on complex pages.

4.1 Memory footprint of mobile browsers

To measure the memory used by different mobile browsers, we open an empty tab and record the PSS[5]. We found the PSS of Chrome, Brave, Microsoft Edge, and Firefox to be 141 MB, 159 MB, 223 MB, and 173 MB, respectively. While Chrome had the least memory usage, Microsoft Edge consumed the most memory. Browsers instantiate and maintain different data structures and use different JavaScript engines (e.g., Chrome uses the V8 JS engine whereas Firefox uses SpiderMonkey) and browser engines[6] (e.g., Chrome uses Blink whereas Firefox uses Gecko as their browser engine), which can result in different memory usage for loaded pages. Therefore, next, we measure the average memory footprint of Alexa top 100 pages when loaded on different browsers.

[4]One could use an online service such as Amazon's Device Farm (http://awsdevicefarm.info/) to use a larger set of mobile devices, however, at the time of writing, Device Farm did not provide remote access to devices with 2 GB or less RAM.

[5]We used Nexus 5 for these experiments with Android 6.0.1 (the highest Android version supported on this phone). along with the following browser versions: Chrome (81.0.4044.17), Firefox (68.7.0), Microsoft Edge (45.02.2.4931), Brave (1.5.131 , Chromium Build 80.0.3987.162).

[6]These are also known as rendering or layout engines.

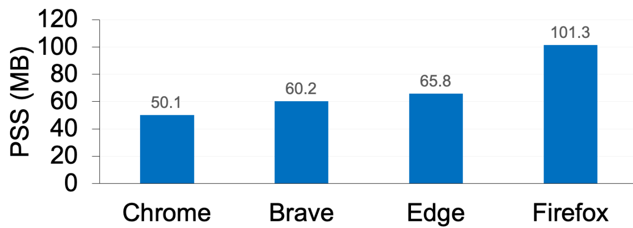


Figure 3: Average memory used by Alexa top 100 Websites when loaded on different mobile browsers. The memory usage is obtained by subtracting the total memory usage of a loaded page (including the browser) from the memory consumed by an empty tab.

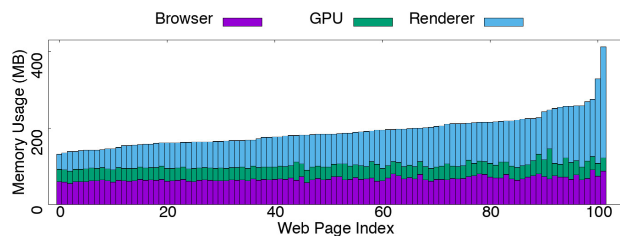


Figure 4: Memory consumed by the Browser, Renderer, and GPU processes for Alexa top 100 Websites (sorted by memory consumed – left to right).

Figure 3 shows that Chrome results in the smallest average memory footprint across all browsers. Interestingly, even though Microsoft Edge consumes the largest memory on an empty page, it induces a smaller average memory footprint for a loaded page than Firefox.

4.2 Deconstructing Chrome's memory usage

The Chrome browser has three main processes: Browser, Renderer, and the GPU process. There is a single Browser process for an instance of Chrome that is responsible for performing the typical functions of a browser (e.g., opening a tab and fetching a Web page). This process then passes the fetched page and its resources to the Renderer process through an IPC mechanism. For each new tab, a separate Renderer process is created, which is responsible for parsing the HTML, building the DOM, CSSOM, and the render tree, running scripts, and painting the contents of a Web page in its respective tab. To perform these tasks the Renderer process uses several engines (e.g., V8 JS engine). The GPU process is used to perform compute-intensive tasks (e.g., compositing). A GPU process does not only use GPU memory but can also take up space in the RAM.

Memory consumption across browser *processes*. Figure 4 shows the memory consumed by the Browser, Renderer, and GPU processes for Alexa top 100 Web pages. Observe

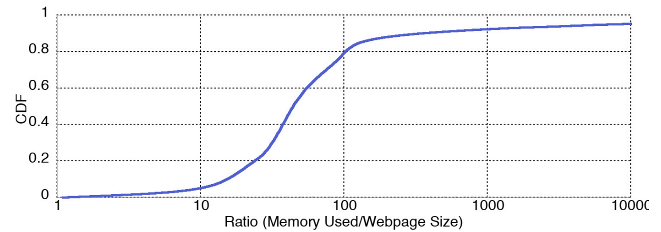


Figure 5: Ratio of the memory footprint of a browser on a loaded page to the Web page size for Alexa top 100 Websites.

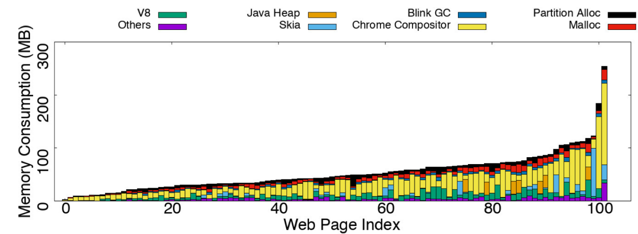


Figure 6: Memory distribution of the Renderer process across browser components for Alexa top 100 Websites.

that the consumed memory ranges from around 130 MB to 410 MB. While the memory consumed by the Browser and GPU processes remain largely the same[7], the primary difference appears in the memory footprint of the Renderer process, which varies from 45 MB to 290 MB. This happens because the Renderer process is responsible for building and maintaining intermediate data structures (e.g., DOM, CSSOM, and the render tree) that consume a significant amount of memory. Note that in Chrome, the CSSOM and DOM trees are combined into a render tree, which is then used to compute the layout of each visible element and serves as an input to the painting process that renders the pixels to screen.

Figure 5 shows the CDF of the ratio of the memory footprint of a loaded Web page (i.e., after subtracting the memory consumed by all processes when only an *empty tab* is open) to its size for Alexa top 100 Web pages. We find that 50% of the Web pages consumed at least 45x more memory than their actual sizes (i.e., the sum of the bytes in the base HTML and all fetched objects) whereas 20% of the pages consumed 100x more memory relative to their sizes. Thus, a page can have orders of magnitude larger memory footprint compared to its size. For example, a 1750 KB Web page in our dataset led to 182 MB of memory allocations in the browser (after excluding the memory taken by an empty tab).

Memory distribution across browser *components*. Next, we examine the distribution of memory occupied by the

[7] The browser process runs the UI and manages tab and plugin processes, thereby performing a similar set of operations for every fetched Web page.

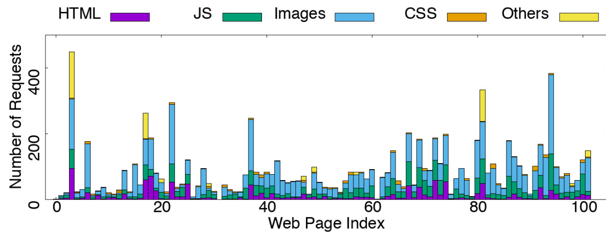


Figure 7: Number of requests made for different objects by Alexa top 100 Web pages (sorted by memory consumed).

Renderer process across different browser components for the same set of Web pages. We find that the compositor[8], V8 (Chrome's JS engine), and Malloc/Partition Alloc[9] take up most memory across most Web pages as shown in Figure 6. This happens because the majority of Web pages, requests for images dominate followed by requests for scripts as shown in Figure 7.

Interestingly, we find that Web pages having large compositor memory footprint may not have a commensurately large number of images. For example, the Web page with the largest memory footprint in our dataset had a smaller number of images than several other Web pages, possibly due to differences in image resolutions and image formats. To understand the impact of image resolutions, image formats, and the number of images on compositor memory, we construct synthetic Web pages. Figure 8 shows that the compositor's memory footprint increases with image resolution across all formats. However, images in GIF and PNG formats contribute to much larger compositor memory, especially for higher resolution images. For example, a 1500x1500 GIF can contribute more than 90 MB to compositor's memory compared to a WebP or JPEG image of the *same* resolution. Next, we fix the image resolution and vary the number of images per Web page. Figure 9 shows that compositor's memory footprint increases with the number of images but the increase varies across formats. For example, for GIFs, the compositor's memory grows much larger for a small number of images but the difference becomes smaller for a large number of images.

Use case 1–*page scrolling*. We now evaluate changes in the memory footprint of a Web page when a user *scrolls* down a page. We find that scrolling can significantly increase the memory footprint of a Web page (e.g., due to rendering of more images in the viewport) as shown in Figure 10. This

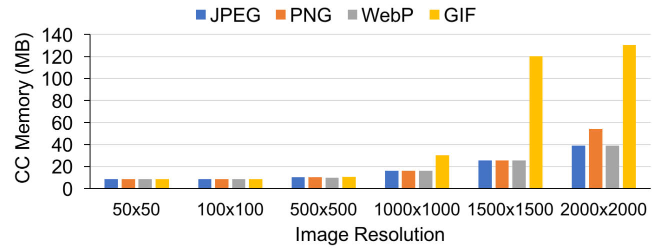


Figure 8: Compositor memory footprint as a function of image resolution.

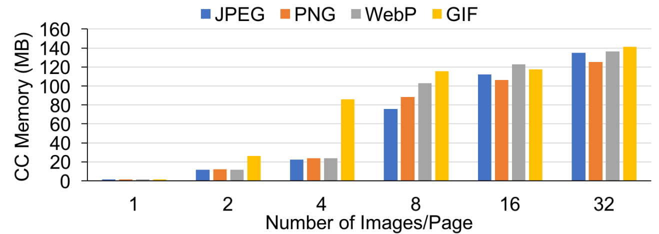


Figure 9: Compositor memory footprint as a function of number of images in a Web page.

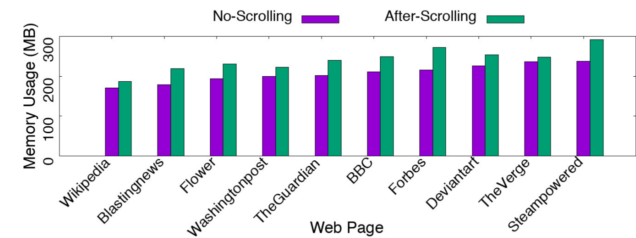


Figure 10: Changes in PSS when a page is scrolled from top to bottom. We consider ten Web pages for this experiment.

can happen both in the presence and absence of lazy loading[10]. For example, bbc.com's PSS increases by more than 50 MB when it is scrolled down. We find that the primary contributor to this increase are the compositor and the JS engine.

Use case 2–*using multiple tabs*. We find that using multiple tabs significantly increases the memory footprint. Figure 11 shows the PSS with one tab, two tabs (one tab being empty), and five tabs (with four tabs being empty). Observe that Chrome's PSS increases by around 38 MB with each empty tab that is open. In Chrome, each tab runs as a separate renderer process and causes memory usage in Java heap and compositor.

In summary, Web pages can consume a large amount of memory on mobile devices even if they have small sizes (e.g., more than 90% of Web pages consumed at least 10x more memory than their sizes). The memory usage can vary widely across use cases and image formats with the compositor and

[8]Compositing refers to the use of multiple backing stores to cache and group chunks of the render tree. It helps in avoiding repainting and makes some features (e.g., scrolling) more efficient.

[9]PartitionAlloc is a memory allocator optimized for security, low allocation latency, good space efficiency.

[10]Lazy loading is technique that defers loading of non-critical resources at page load time.

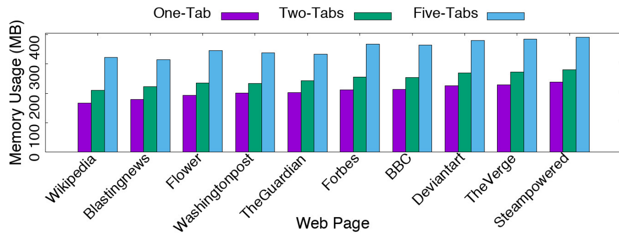


Figure 11: Changes in PSS when a Web page is loaded in case of 1, 2, and 5 tabs scrolled from top to bottom. We consider ten Web pages for this experiment.

the JS engine contributing most to memory in the majority of cases.

4.3 Memory usage across Android Versions

We now evaluate the impact of using different Android versions on Chrome's memory usage. Using the same Chrome version, we load Alexa top 100 Web pages on three Android versions: Android 6 (Marshmallow), Android 7 (Nougat), and Android 8 (Oreo). We found that the memory footprint of Web pages increased with newer versions of Android but not substantially. In particular, we found the average memory footprint of Alexa top 100 Web pages for Marshmallow, Nougat, and Oreo to be 49.8 MB, 57.4 MB, and 59.6 MB, respectively[11]. We conjecture that the increase in memory is at least due to two reasons: (i) Android 7 introduced a new API for multitasking (MultiInstanceManager) [2], which is utilized by Chrome in Android versions 7, and above and (ii) the use of different themes and window properties for across Android versions. We measure the memory overhead of the multitasking API by creating an instance of the MultiInstanceManager class in a blank android application and found it to be approximately 7-8 MB.

5 IMPACT OF MEMORY PRESSURE

When the memory usage of applications grows or there are too many applications running in the system, the system memory may become too low for a device to operate efficiently. In such scenarios, Android makes use of two system daemons: *kswapd* and *lmkd*. In particular, when the number of *free* pages become too low, *kswapd* kicks in and starts reclaiming *cached* pages to find more free memory. If *kswapd* is unable to reclaim any cached pages, the device can start to thrash. Thus in Android, the low memory killer daemon *lmkd* is started when the numbers of cached pages get too low[12]. To keep system performance at acceptable levels, *lmkd* starts killing processes in order of their priorities (starting from the lowest priority first as shown in Table 4) to find more free memory [7].

Process State	Type of Process
Native	daemons (e.g., kswapd, logd, adbd)
System	system services (e.g., system_server)
Persistent	persistent apps (e.g., telephony, WiFi)
Foreground	contains foreground activity
Visible	contains activities that are visible
Perceptible	(e.g. background music playback)
Service	contains an application service
Home	contains the home application
Previous	the previous foreground application
Cached	cached, thus can be killed without any disruption

Table 4: Android process list in order of priority (*native processes* have the highest priority whereas *cached* processes have the *lowest priority*).

Thus, there are two important consequences for a device operating in low memory regimes:

- **An application can *crash* or get killed by *lmkd*.** Since *lmkd* kills processes based on their priority to free up memory, in case there are no more low priority processes left to kill to free more memory, *lmkd* can kill the foreground application as well.
- **Application performance can degrade**. This can happen for three reasons: (a) Due to interference with native processes (i.e., *kswapd* and *lmkd*) that can take up CPU cycles and are strictly prioritized over foreground processes, (b) when a process tries to allocate memory but there is no free memory available, the kernel blocks the allocation while it frees up a page. This often requires waiting for *lmkd* to kill a process or disk I/O to flush out a storage-backed page, which increases delay and (c) if *kswapd* reclaims cached pages of the *foreground* app, and later the apps needs the data, it will need to fetch it from device storage, which may cause noticeable delay and degrade app performance.

Defining *Memory Pressure.* In Android, the memory usage on a device is captured through *memory pressure* (P), which is defined as:

$$P = (1 - R/S) * 100 \quad (1)$$

where R and S are the number of reclaimed and number of scanned pages, respectively. Note that P is measured over a window, which is the number of scanned pages after which

[11] We observed that Nougat and Oreo keep a larger amount of free memory, which is maintained by the variable min_free_kbytes. In particular, while Marshmallow sets it to 4893 kB, Nougat and Oreo set it to 5168 kB. We set these values the same for all versions in our experiments to measure the impact of other aspects of Android.

[12] Note that each Android app runs in a separate Dalvik VM, which locally runs a garbage collector (GC) to reclaim any app memory not being used. GC runs when the app's heap size grows large independent of whether the *system* is operating under low memory or not. However, when GC daemon runs, it can pause app execution, which may impact performance [21].

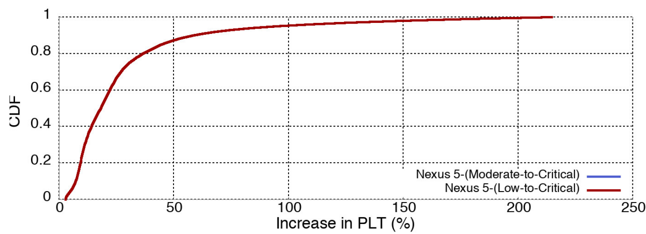


Figure 12: Web page loads under different memory pressure states across Alexa top 50 pages.

P is computed[13]. If most pages can be reclaimed, P would be low. However, if the number of cached pages decrease, this will lead to high memory pressure regimes signifying that the system is running under low memory. In Android, memory pressure is referred to as "low" when $P \leq 60$, "medium" (or moderate) when $60 < P < 95$, and "critical" (or high) when $P \geq 95$ [5, 28].

5.1 Impact on Web Browsing

Impact on PLTs and crashes. We now analyze the impact of memory pressure on Web browsing. We introduce memory pressure via a custom (native) Android app that we built, which allocates memory until a target memory pressure regime is achieved. For example, to induce critical memory pressure, we continue to allocate memory until we start receiving critical pressure signals from the kernel. We only start page loads after having achieved a given memory pressure regime.

Figure 12 shows the increase in PLT when the device operates under critical memory pressure compared to moderate and low memory pressure states for Alexa top 50 pages. We find that the average PLT increases by 28.2% under critical memory pressure. This happens because under low memory *kswapd* starts reclaiming cached pages during the page load process. Since *kswapd* has native priority and is strictly prioritized over foreground apps, it takes away CPU cycles from Chrome processes.

We find that the impact is the greatest on Web pages with a large memory footprint (e.g., 5% of the Web pages experience more than 100% increase in their in PLT) because for such pages the chances are higher that the system runs out of free memory under high memory pressures. In such cases, *lmkd* will start killing processes leading to larger waiting times for more free memory to become available, which increases PLTs. In our evaluation, We found that about 6.3% of the Web pages crashed under high memory pressure regimes.

Impact of number of cores. Next we analyze the impact of CPU cores under different memory pressures on the performance of Web browsing. We randomly pick five Web pages

[13]Currently, the window size is set to 512 pages (or 2MB for 4KB pages).

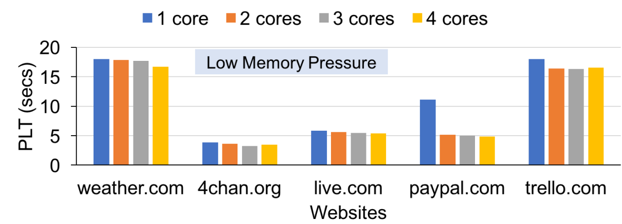


Figure 13: PLT as a function of CPU cores for five randomly selected Web pages from Alexa top 100 under low memory pressure.

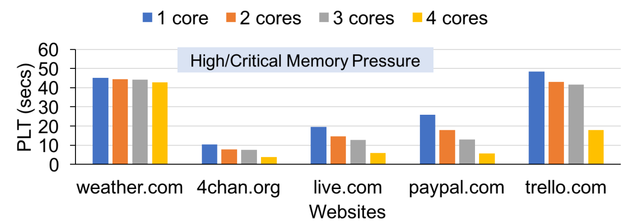


Figure 14: PLT as a function of CPU cores for five randomly selected Web pages from Alexa top 100 under high/critical memory pressure.

from Alexa top 100 and measure their PLTs under low and high/critical memory pressure regimes. Figure 13 shows that 1-2 CPU cores are enough for achieving high performance and using more cores does not improve performance commensurately. These results match the observations made by Dasari et al. [35]. Next, we introduce high memory pressure and then load pages with varying number of cores. Figure 14 shows that increasing the number of CPU cores improves PLT significantly. In particular, using 4 cores as opposed 2 cores improves PLTs by 51%-69% for four Web pages (i.e., 4chan.org, live.com, paypal.com, and trello.com). This happens because high priority system daemons *kswapd* and *lmkd* consume significant CPU cycles and can also switch between CPU cores.

Impact of mobile devices and Android versions. Next, we evaluate page load performance over three mobile devices (i.e., Nexus 5, Nexus 5X, and Nexus 6P) and three Android versions (Marshmallow, Nougat, and Oreo). We make three key observations from our results: (i) Across *each* mobile device and Android version we considered, the PLTs were (19.6%-29%) higher under high/critical pressure compared to when the device is operating under low memory pressure as shown in Figure 15, (ii) Shifting from Nougat to Oreo (i.e., to a more recent Android version) decreased the average PLT by 2.8%, which suggests better handling of memory pressure situations in more recent Android versions and (iii) Using Oreo on Nexus 6P resulted in an increase in the average PLT by 9.4% under high memory pressured compared Nexus 5X despite the former having 8 cores and 3 GB of RAM compared

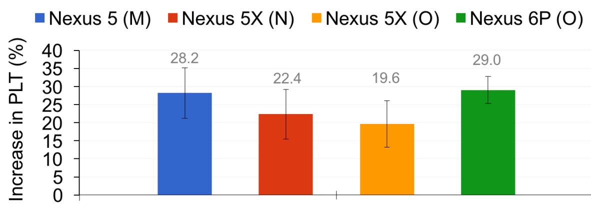


Figure 15: Increase in PLT under high/critical memory pressure across three mobile devices and three Android versions for Alexa top 50 pages.

to 6 cores and 2 GB RAM in Nexus 5X. We conjecture this is due to device-specific optimizations in Android for low-RAM devices [6] (e.g., turning off memory-intensive application features, keeping a smaller *kswapd* threshold on free memory on low-RAM devices).

6 OPTIMIZATIONS FOR LOW MEMORY

A typical Web page can have different types of objects (e.g., JS, CSS, images and iframes). Our study of Alexa top 100 Web pages showed that images and JS contribute the most to the memory footprint of Web pages on mobile browsers. Thus, to reduce the memory footprint of Web pages to better handle high memory pressure scenarios, we propose the following optimizations.

(A) Debloating JS. Web pages commonly use JS libraries (e.g., jQuery, AngularJS, React) for ease of development. These libraries are often bloated and can have a large memory footprint. Thus, we first analyze how common are different JS libraries on popular Web pages. Figure 16a shows the usage of (five most popular) JS libraries across Alexa top 880 Web pages. We find that jQuery is the most widely used JS library on Web pages followed by React, Modernizr, Moment_js, and AngularJS.

Next, we analyze the memory overhead induced by JS libraries when reading, inserting, and updating the DOM. To this end, we consider six operations involving a table (e.g., creating 100 rows, updating styles, swapping rows, and finally clearing rows) and implement them in jQuery, AngularJS, React, and VanillaJS (i.e., plain JS that does not use any external JS library). Figure 16b shows the Java heap size across different operations. Observe that using VanillaJS results in the smallest memory footprint across all operations. In particular, VanillaJS results in 1.4x-1.9x smaller JS heap size compared to jQuery, 1.6x-12.3x smaller compared to AngularJS, and 1.1x-9.8x smaller than React across all operations.

We also measure the scripting/execution time of these operations and find that VanillaJS takes the least amount of time across *all* operations. Compared to jQuery, VanillaJS provides 3x-105x smaller scripting times across operations. These results suggest that when developing simple pages, avoiding the use of JS libraries can improve memory footprint as well as reduce scripting times.

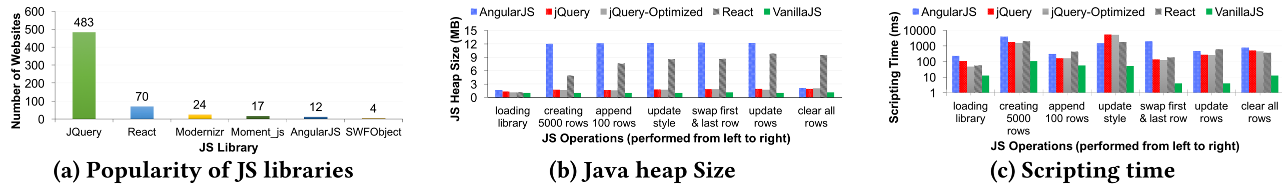
jQuery-Optimized. In this version of jQuery, we remove unused functions from the library by examining the Web page. This decreases (a) Java heap size (by ~1 MB) as well as the (b) loading time of the library (by 100 ms) for two reasons: (a) this results in less code to parse and compile for the JS engine, V8 and (b) there are less or smaller internal data structures the library now maintains.

(B) Choosing memory efficient image formats. To assess possible improvements that can be brought about by changing image formats, we first analyze how *common* are different image formats in popular Web Pages. Figure 18 shows the usage of image formats across 43309 images in Alex top 880 Web pages. Observe that GIF (32%), JPEG (30%), and PNG (23%) are the three dominant image formats followed by WebP (6%).

Next, we carry out a controlled experiment by loading a synthetic page with 16 images. We encode all images in a one format and measure their memory footprint and then repeat this process for other formats. Figure 19 shows the PSS of the renderer process and its memory distribution across three major components for different image formats. Observe that the PSS varies widely across image formats. In particular, GIF results in the largest PSS (i.e., 279 MB), followed by PNG (i.e., 227 MB) whereas JPEG (162 MB) and Google's WebP (167 MB) format result in the smallest memory footprint.

Interestingly, we find that while the compositor (cc) memory footprint does not vary significantly across formats (107 MB-123 MB) in this case[14], the difference is due to Webcache – which shows the actual size of images – and malloc, which reflects the memory consumed by data structures when an image is loaded. The difference in the memory consumption grows larger with the number of images in a Web page (see Figure 17. This happens because GIF and PNG both use lossless compression but the latter usually delivers better compression. Moreover, GIF limits the color palette (i.e, collection of colors that an image uses) to at most 256 colors wheres PNG allows for larger number of palettes and thus produces high quality images but at the cost of a higher file size than other formats. JPEG uses a combination of lossy and lossless optimization to reduce file size of the image asset. Lossless WebP compression uses already seen image fragments in order to exactly reconstruct new pixels whereas lossy WebP compression uses predictive coding to encode an image.

[14]The difference in compositor memory footprint does vary significantly in case of high resolution images or when the number of images/page are small as discussed in Section 4.



(a) Popularity of JS libraries (b) Java heap Size (c) Scripting time

Figure 16: (a) Popularity of JS libraries in Alexa top 880 Websites (we list top five), (b) Java heap size for several operations performed by different JS libraries, and (c) time to perform different operations.

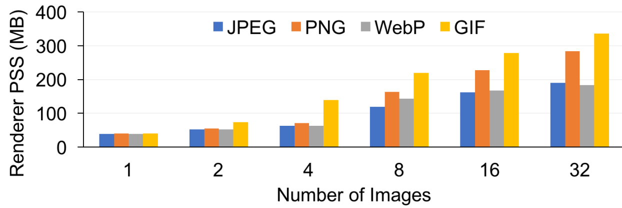


Figure 17: PSS of the renderer process as we increase the number of images in a Web page.

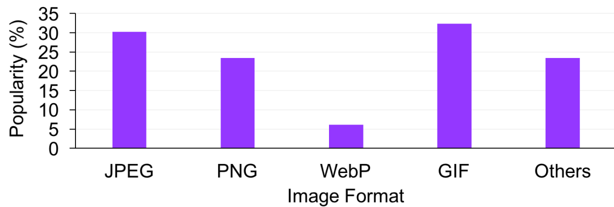


Figure 18: Popularity of image formats in Alexa top 880 Web pages.

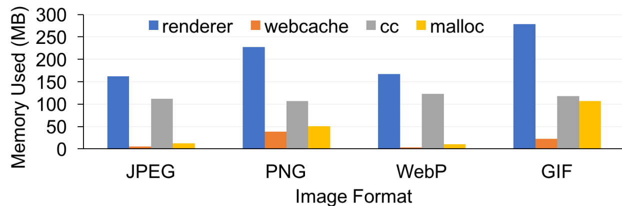


Figure 19: PSS of the renderer process and three browser components (i.e., Webcache, cc, and malloc) for different image formats. For this experiment, we used a synthetic page having just 16 images.

Thus, Websites with several images can significantly reduce their memory footprint on users' devices if they opt for image formats such as JPEG and WebP and serve low resolution images.

7 DISCUSSION AND IMPLICATIONS

Our finding have implications for different stakeholders in the smartphone ecosystem.

- *Website Developers:* Our work provides insights for website developers for reducing the memory footprint of their Web pages, which can speed up page loads when a mobile device is running low on memory or have a small RAM size to begin with. For example, using native JS, debloated JS libraries, and memory-efficient image formats can reduce memory pressure and improve scripting times.
- *Original Equipment Manufacturers:* For OEMs, our work provides insights about mobile hardware features that impact Web browsing under high memory pressure. For example, having more CPU cores (e.g., 4 compared to 1 or 2) can significantly improve Web browsing performance under high memory pressure scenarios due to reduced interference with high priority system daemons such as *kswapd*.
- *Application Users:* Crashes and poor application performance frustrate users. Our work can inform users about scenarios under which they can experience slow page loads and how user actions (e.g., number of open apps) impact application behaviour.
- *System Developers:* Our work sheds light on possible system-level mechanisms that can help improve performance (e.g., *kswapd* scheduling). For example, careful assignment or cores to system daemons can reduce interference with foreground processes.

8 RELATED WORK

There is a large body of work on optimizing mobile Web performance, including design and optimization of proxy services (e.g., [20, 32, 41, 42]), reducing the overhead of client-side JavaScript (e.g., [39, 45]), prefetching [15, 34], understanding dependencies in the page load process [46], studying the impact of network infrastructure [47], and analysis of mobile devices in developing countries [33, 35].

Analysis of page load bottlenecks. WProf [43] is a lightweight in-browser profiler that produces a detailed dependency graph of the activities that make up a page load. In [38], authors perform comparison between mobile and non-mobile browsers and characterize performance bottlenecks. They found that computation activities, and not the network transfer times, are the main bottleneck on mobile browsers. In contrast, our work focuses on understanding the *memory* footprint of Web pages and page load performance of Web browsing when a device is operating low on memory. Ahmad et al. [33] analyzed the characteristics of

mobile devices in a developing region and observed that device-level bottlenecks are highly likely.

Proxy and backend accelerators/frameworks. FlyWheel [32] is an HTTP proxy service that reduces data for mobile users by compressing responses in-flight between origin servers and browsers. However, FlyWheel does not provide support for HTTPS pages. The Opera Mini [19] browser works by offloading compute intensive tasks, such as scripting, to a proxy service. However, it often comes at the cost of interactivity (e.g., OperaMini does not support touch events). Some recent approaches, such as Prophecy [39] and Shandian [46], return post-processed versions of objects to reduce client-side computation and bandwidth costs. For example, in Prophecy, Web servers precompute the JavaScript heap and the DOM tree for a page and when a mobile browser requests the page, the server returns a write log that contains a single write per JavaScript variable or DOM node. AMP accelerates mobile page loads by requiring pages to be written in a restricted dialect of HTML, CSS, and JavaScript that is faster to load [1, 37].

Server push systems. Several systems aim to accelerate mobile page loads by leveraging HTTP/2's server push feature, where servers proactively push resources to clients in anticipation of future requests [36, 40, 44]. While useful, such approaches can increase memory pressure and thus impact user-perceived performance especially on low-end devices [15].

Solutions for entry-level devices. Several ongoing and recent efforts aim to improve mobile QoE on entry-level devices [4, 10, 42]. Free Basics is a Facebook initiative to provide zero-rated Web services in developing countries. The Free Basics platform is available in over 60 countries that offers a collection of Web services that serve light pages to users so that they can load faster on low-end mobile devices while saving data. Free Basics services do not support JavaScript, large images, or videos. Google's Web Light service [42] is a proxy-based service available in several developing countries, which transcodes pages into pruned versions and improves page load performance at the cost page quality and site interactivity. Android Go is a pruned version of Android designed for entry-level devices [4].

9 CONCLUSION

In this work, we analyzed the impact of device memory usage on mobile browsing performance over different mobile phones, mobile browsers, Android versions, and across different application use cases. Our analysis of popular Web pages shows that their memory footprint can be orders of magnitude larger than their corresponding page sizes. This can often lead to slow browsing performance or page crashes when a device is running low on memory or if a device has small memory to begin with. We find that debloating JS and using memory-efficient image formats can significantly reduce the memory footprint of Web pages and improve load performance. We view this work as an initial step towards a better understanding of how a device running under low memory impacts application performance.

10 ACKNOWLEDGEMENT

We are grateful to the SIGCOMM CCR reviewers for their insightful comments and suggestions. This work was partially funded by a Google Faculty Research Award and LUMS FIF grant.

REFERENCES

[1] [n. d.]. AMP: Building the future web, together. ([n. d.]). https://www.ampproject.org/.
[2] [n. d.]. Android 7.0 for Developers. ([n. d.]). https://developer.android.com/about/versions/nougat/android-7.0.
[3] [n. d.]. Android: Crashes. ([n. d.]). https://developer.android.com/topic/performance/vitals/crash.
[4] [n. d.]. Android Go. ([n. d.]). https://www.android.com/versions/go-edition/.
[5] [n. d.]. Android: LMKD in Userspace. ([n. d.]). https://source.android.com/devices/tech/perf/lmkd.
[6] [n. d.]. Android: Low RAM Configuration. ([n. d.]). https://source.android.com/devices/tech/perf/low-ram.
[7] [n. d.]. Android: Processes and Application Lifecycle. ([n. d.]). https://developer.android.com/guide/components/activities/process-lifecycle.
[8] [n. d.]. Brave browser moves to Chromium codebase. ([n. d.]). https://zd.net/3cZHDp7.
[9] [n. d.]. Build for Android (Go edition): optimize your app for global markets (Google I/O '18). ([n. d.]). https://bit.ly/2UKLQDl.
[10] [n. d.]. Free Basics Platform. ([n. d.]). https://developers.facebook.com/docs/internet-org.
[11] [n. d.]. Google Play Store. ([n. d.]). https://developer.android.com/distribute/google-play.
[12] [n. d.]. How to fix low memory and low storage issues on Android. ([n. d.]). https://bit.ly/2Hc8Mre.
[13] [n. d.]. http archive. ([n. d.]). https://httparchive.org/.
[14] [n. d.]. ICT Facts and Figures. ([n. d.]). https://www.itu.int/en/ITU-D/Statistics/Documents/facts/ICTFactsFigures2017.pdf.
[15] [n. d.]. Introducing NoState Prefetch. ([n. d.]). https://developers.google.com/web/updates/2018/07/nostate-prefetch.
[16] [n. d.]. Kiwi Browser - Fast and Quiet. ([n. d.]). https://bit.ly/2WZAIH2.
[17] [n. d.]. Low memory on Android: how to fix it. ([n. d.]). https://bit.ly/2Mn73U3.
[18] [n. d.]. Microsoft Edge. ([n. d.]). https://www.microsoftedgeinsider.com/en-us/.
[19] [n. d.]. Opera mini browser. ([n. d.]). http://www.opera.com/mobile/.
[20] [n. d.]. Opera Turbo. ([n. d.]). http://www.opera.com/turbo.
[21] [n. d.]. Overview of memory management. ([n. d.]). https://developer.android.com/topic/performance/memory-overview.
[22] [n. d.]. Speed Index. ([n. d.]). https://web.dev/speed-index/.
[23] [n. d.]. Telemetry. ([n. d.]). https://chromium.googlesource.com/catapult/+/HEAD/telemetry/.

[24] [n. d.]. The most popular Android smartphones in 2019. ([n. d.]). https://deviceatlas.com/blog/most-popular-android-smartphone.
[25] [n. d.]. The Most Popular Browsers. ([n. d.]). https://www.w3schools.com/browsers/.
[26] [n. d.]. Vivaldi. ([n. d.]). https://en.wikipedia.org/wiki/Opera_(web_browser).
[27] [n. d.]. Vivaldi. ([n. d.]). https://vivaldi.com/.
[28] [n. d.]. vmpressure.c. ([n. d.]). https://android.googlesource.com/kernel/msm/+/android-9.0.0_r0.31/mm/vmpressure.c.
[29] [n. d.]. Web Light: Faster and lighter mobile pages from search. ([n. d.]). https://support.google.com/webmasters/answer/6211428?hl=en.
[30] [n. d.]. Web Page Replay. ([n. d.]). https://github.com/catapult-project/catapult/blob/master/web_page_replay_go/README.md.
[31] [n. d.]. WebPageTest. ([n. d.]). https://www.webpagetest.org/.
[32] Victor Agababov, Michael Buettner, Victor Chudnovsky, Mark Cogan, Ben Greenstein, Shane McDaniel, Michael Piatek, Colin Scott, Matt Welsh, and Bolian Yin. 2015. Flywheel: Google's Data Compression Proxy for the Mobile Web. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*.
[33] Sohaib Ahmad, Abdul Lateef Haamid, Zafar Ayyub Qazi, Zhenyu Zhou, Theophilus Benson, and Ihsan Ayyub Qazi. 2016. A View from the Other Side: Understanding Mobile Phone Characteristics in the Developing World. In *Proceedings of the 2016 Internet Measurement Conference (IMC '16)*. 319–325. https://doi.org/10.1145/2987443.2987470
[34] Michael Butkiewicz, Daimeng Wang, Zhe Wu, Harsha V. Madhyastha, and Vyas Sekar. 2015. KLOTSKI: Reprioritizing Web Content to Improve User Experience on Mobile Devices. In *NSDI*.
[35] Mallesham Dasari, Santiago Vargas, Arani Bhattacharya, Aruna Balasubramanian, Samir R. Das, and Michael Ferdman. 2018. Impact of Device Performance on Mobile Internet QoE. In *Proceedings of the Internet Measurement Conference 2018 (IMC '18)*. 1–7. https://doi.org/10.1145/3278532.3278533
[36] Jeffrey Erman, Vijay Gopalakrishnan, Rittwik Jana, and K. K. Ramakrishnan. 2013. Towards a SPDY'ier Mobile Web?. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '13)*. 303–314. https://doi.org/10.1145/2535372.2535399
[37] Byungjin Jun, Fabian E. Bustamante, Sung Yoon Whang, and Zachary S. Bischof. 2019. AMP Up Your Mobile Web Experience: Characterizing the Impact of Google's Accelerated Mobile Project. In *The 25th Annual International Conference on Mobile Computing and Networking (MobiCom '19)*. Article 4, 14 pages. https://doi.org/10.1145/3300061.3300137
[38] Javad Nejati and Aruna Balasubramanian. 2016. An In-Depth Study of Mobile Browser Performance. In *Proceedings of the 25th International Conference on World Wide Web (WWW '16)*. 1305–1315. https://doi.org/10.1145/2872427.2883014
[39] Ravi Netravali and James Mickens. 2018. Prophecy: Accelerating Mobile Page Loads Using Final-State Write Logs. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. 249–266.
[40] Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. 2017. Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. 390–403. https://doi.org/10.1145/3098822.3098851
[41] Shailendra Singh, Harsha V. Madhyastha, Srikanth V. Krishnamurthy, and Ramesh Govindan. 2015. FlexiWeb: Network-Aware Compaction for Accelerating Mobile Web Transfers. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom '15)*. 604–616. https://doi.org/10.1145/2789168.2790128
[42] Ammar Tahir, Muhammad Tahir Munir, Shaiq Munir Malik, Zafar Ayyub Qazi, and Ihsan Ayyub Qazi. 2020. Deconstructing Google's Web Light Service. In *Proceedings of The Web Conference 2020 (WWW '20)*. 884–893. https://doi.org/10.1145/3366423.3380168
[43] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. 473–486.
[44] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2014. How Speedy is SPDY?. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. 387–399.
[45] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *NSDI*.
[46] Xiao Sophia Wang, Arvind Krishnamurthy, and David Wetherall. 2016. Speeding up Web Page Loads with Shandian. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. 109–122.
[47] Yasir Zaki, Jay Chen, Thomas Pötsch, Talal Ahmad, and Lakshminarayanan Subramanian. 2014. Dissecting Web Latency in Ghana. In *Proceedings of the 2014 Conference on Internet Measurement Conference (IMC '14)*. 241–248. https://doi.org/10.1145/2663716.2663748

A APPENDIX: CODE AND DATA

To replicate the experimental results reported in this work, we have created a public repository[15]. In this repository, we have provide all the code and data needed to reproduce the experiments described in this paper. This includes the tools we developed to the run the experiments, scripts for analysis and plotting of the generated data. In addition, we also provide information on how we setup third party tools we used for our experiments. If additional help is needed, please contact any of the co-authors.

For our experiments, we use the userdebug build in Android. We track the memory footprint of Chrome through Google's Telemetry tool [23]. For measuring Web page sizes and gathering other page specific statistics, we use WebPageTest, an open-source tool for measuring Web performance [31]. For inducing memory pressure on a mobile, we developed a custom Android application using SDK and NDK. To use this application, one needs to root the device. For tracking PSS of applications, we developed another custom Android application.

All the experimental data and code is in the repository folder *web_browser_experiments*. The main folder is further divided into three sub-folders: *am_footprint_web*, *mp_simulation_web* and *base_for_web_optimization*. These folders provide information on reproducing results corresponding to specific sections of the paper. Below we provide a description of the contents of these sub-folders.

A.1 Application Memory Footprint

The *am_footprint_web* folder contains the link to the Telemetry archive, which is the primary tool we have used for Web tracing (instructions to run Telemetry are also provided

[15] https://github.com/nsgLUMS/mobileLowMem

there). Another folder named *user_debug_build_images* contains the userdebug builds for the devices we used in our experiments. To run Telemetry, a device must be flashed with the userdebug build of Android. After completing the flashing and setting up Telemetry on a local machine, one can easily run Telemetry benchmarking on a device. There are many benchmarks available in Telemetry but for our use case, we have used *memory_top_10_mobile* to the component-wise distribution of memory. We wrote scripts to parse CSV files generated by Telemetry. These scripts are available in the folder named *csv_parsers*. Figures 4, 5 and 6 can be drawn from the final parsed CSV. We also have shared a file containing URLs of the *Alexa_top_100* Web pages we used in our experiments under the *samples* folder. The folder of *web_page_test* contains instructions about how to use and extract results of WebPageTest tests (e.g., number of requests generated by every page, number of objects downloaded, etc), which were used to generate Figure 7.

A.2 Memory Pressure

The folder *mp_simulation_web* contains four sub-folders: *mp_simulator_apk*, *root_images*, *impact_on_devices*, and *core_binding*. In the *mp_simulator_apk* folder, there is an APK for Android devices – which we used for generating memory pressure on the device – along with the instructions about how to run that application and how to track memory pressure using the Android Profiler of ADB logcat. As our application requires root permissions, the target device must be rooted, for which we also have included root images in the *root_images* folder. By using the *mp_simulator* application, we introduce the desired memory pressure and then run Telemetry as discussed earlier but this time we use the *loading_mobile* benchmark to get component-wise time division of Web page loading until the onload event. In the folder of *core_binding*, we have mentioned how to bind cores to the process and then run Telemetry to observe its impact on a Web page's load time. The folder *mp_simulation_web* contains experimental results in the form CSV files that can be used to generate Figure 15.

A.3 Optimizations

The data and instructions to perform experiments related to optimizations for Web browsing are in the folder *base_for_web_optimizations*. The *samples* folder inside the parent directory contains further two sub-folders: *images* and *synthetic_pages*. To obtain memory footprints of different browser components, we used Telemetry same as described in A.1.