Here is the result of analyzing your file(s) with TeXtidote. Hover the mouse over highlighted portions of the document to read a tooltip that gives you some writing advice.

Found 746 warning(s)

```
1
2  \documentclass{kththesis}
3
4  \usepackage{graphicx}
5  \usepackage{todonotes}
6  \usepackage{blindtext} % This is just to get some nonsense text in this template,
   can be safely removed
7  \usepackage{hyperref}
8  \usepackage{subcaption}
9  \usepackage{array}
10 \usepackage{enumitem}
11 \usepackage[boxruled,vlined, figure]{algorithm2e}
12
13 \usepackage{xcolor}
14
15 \usepackage{csquotes} % Recommended by biblatex
16 \usepackage[style=numeric,sorting=none,backend=biber]{biblatex}
17 \addbibresource{references.bib} % The file containing our references, in BibTeX
   format
18 \title{Comparison of Smoothness in Progressive Web Apps and Mobile Applications on
   Android}
19 \alttitle{Jämförelse av smidighet i progressiva webbappar och mobilapplikationer på
   Android}
20 \author{Camille Fournier}
21 \email{camillef@kth.se}
22 \supervisor{Javier Cabrera-Arteaga}
23 \examiner{Benoit Baudry}
24 \hostcompany{Odiwi} % Remove this line if the project was not done at a host company
25 \programme{Master in Computer Science}
26 \school{School of Electrical Engineering and Computer Science}
27 \date{Spring 2020}
28
29 % Uncomment the next line to include cover generated at https://intra.kth.se/kth-
   cover?l=en
30 % \kthcover{kth-cover.pdf}
31
32 % Commands
33 \newcommand{\citationneeded}{\todo{Citation needed}[!]}
34 \newcommand{\red}[1]{{\color{red} #1 }}
35
36
37
38 \newcommand*\badge[1]{
39 \tikz[baseline=(char.base)]{
40 \node[shape=circle,text=black,draw=black,fill=white,inner sep=2pt] (char) {#1};}
41 }
42
43 \begin{document}
44 \sloppy % prevent non cut in large url
```

```
45
46
47 % Frontmatter includes the titlepage, abstracts and table-of-contents
48 \frontmatter
49 %\todo[color=magenta]{Centered title may be better}
50
51 \titlepage
52
53 \begin{abstract}
54 One of the main challenges of mobile development lies in the high fragmentation of
   mobile platforms. Developers often need to develop the same application several
   times for all targeted platforms, raising the cost of development and maintenance.
   One solution to this problem is cross-platform development, which traditionally only
   includes mobile applications. However, a new approach introduced by Google in 2015
   also includes web applications. Progressive Web Apps, as they are called, are web
   applications that can be installed on mobile and behave like mobile applications.
   This research aims at studying and comparing their performance to mobile
   applications on Android, especially in terms of smoothness, memory and CPU usage. To
   that end, we analyzed the Rendering pipeline of Android and Chrome and deducted a
   smoothness metric. Then, a Progressive Web App, a Native Android and a React Native
   Interpreted Application were developed and their performance measured in several
   scenarios. The results imply that Progressive Web Applications, though they have
   great benefits, are not as smooth as Mobile applications on Android. Their memory
   performance and CPU usage lag behind Native Applications, but are similar to
   Interpreted applications.
55 \end{abstract}
56
57
58 \begin{otherlanguage}{swedish}
59 \begin{abstract}
60 En av de största utmaningarna med mobilutveckling ligger i den höga
   fragmenteringen av mobilplattformar. Utvecklare behöver ofta utveckla samma
   applikation flera gånger för alla riktade plattformar, vilket ökar
   kostnaderna för utveckling och underhåll. En lösning på detta problem är
   plattformsutveckling, som traditionellt endast innehåller mobilapplikationer.
   En ny metod som Google introducerade 2015 inkluderar dock webbapplikationer.
   Progressiva webbappar, som de kallas, är webbapplikationer som kan
   installeras på mobil och bete sig som mobilapplikationer. Denna forskning
   syftar till att studera och jämföra deras prestanda med mobila applikationer
   på Android, särskilt när det gäller smidighet. För detta ändamål
   analyserade vi Rendering-pipeline för Android och Chrome och drog en
   jämnhetsmetrisk. Sedan utvecklades en progressiv webbapp, en Native Android och
   en React Native Hybrid-applikation och deras prestanda uppmättes i flera
   scenarier. Resultaten antyder att progressiva webbapplikationer, även om de
   har stora fördelar, inte är lika smidiga som mobilapplikationer på Android.
61 \end{abstract}
62 \end{otherlanguage}
63
64 \section*{Acknowledgement}
65 \paragraph{}
66 Before presenting my thesis, I would like to thank the people who helped me conduct
   this research. First, I want to thank Cristian Bogdan who helped me choose a
   Master Thesis project and my examinator Benoit Baudry who helped me define it.
67 Then, I want to thank the company Odiwi that welcomed me and helped me conduct my
   research with their equipment. Last but not least, I want to thank my academic
   supervisor, Javier Cabrera Arteaga, who helped me and supported me a lot during
   this project.
68
69 \tableofcontents
70
71
72 % Mainmatter is where the actual contents of the thesis goes
73 \mainmatter
74
75
```

```
\chapter{Introduction}

\indent

One of the biggest challenges of mobile development today lies in the high fragmentation of mobile platforms \cite{MobileDevChallenges} with developers building the same app over multiple platforms. A popular solution is cross-platform development, that is to say developing with a framework capable of using the same code for multiple platform builds. However, such cross-platform development does not concern web applications that developers still have to develop separately from mobile applications. To answer this problem, Google introduced in 2015 the concept of \textit{Progressive Web Apps} \cite{PWA_intro}, a term first coined by designer Frances Berriman and Google Chrome developer Alex Russel \cite{PWA_blog, PWApossibleUnifer}.


\paragraph{}
Progressive Web Apps, also called PWA, are Web Applications that can be installed on a mobile phone by the browser and provide an offline experience. End users may even think of them as real mobile applications as they are displayed in fullscreen and can be detected by the Application Manager depending on the device and the browser.
Progressive Web Apps have become more and more popular since they were introduced to the application development community. Several success stories can attest it : AliBaba\footnote{https://developers.google.com/web/showcase/2016/alibaba} and Aliexpress\footnote{https://developers.google.com/web/showcase/2016/aliexpress} reported an increase of respectively 76\% and 104\% of their conversion rate after releasing their PWA. Twitter\footnote{https://medium.com/@paularmstrong/twitter-lite-and-high-performance-react-progressive-web-apps-at-scale-d28a00e780a3} saw a reduction of their data usage coupled with a 75\% increase of tweets sent. Pinterest\footnote{https://medium.com/dev-channel/a-pinterest-progressive-web-app-performance-case-study-3bd6ed2e6154} reported a 40\% increase of user time and a 44\% of user-generated revenue.

\paragraph{}
Existing research suggests that PWA are a good alternative to mobile applications. Their installation size and energy consumption are smaller \cite{PWAapplicability}. Their launch time \cite{Biorn-Hansen2} and their First paint metric\footnote{Time to render the first pixel}\cite{PWAapplicability} can be lower than a mobile application. For a developer, they are not much more complex than regular web applications to develop\cite{JohannsenFabian2018PWAa} and easier to maintain than native applications. \newline

However, no research has been conducted regarding the run-time performance of Progressive Web Applications, especially the performance of the User Interface, which can be evaluated with two key elements: the responsiveness and the smoothness. The responsiveness refers to how fast the application can respond to user input. The smoothness refers to how fast the application render new frames, that is how fast the application can draw new content on the screen. The latter can impact the former as applications usually respond to user input visually and need to render new frames. Other important factors of the run-time performance are CPU usage and memory performance as those resources are limited on mobile phones. At the time of writing, there was no study that examined those performance parameters in Progressive Web Apps.

\paragraph{}
This thesis will focus on several aspects of the run-time performance of Progressive Web Apps and aim at answering the following questions:

\begin{itemize}
\item \textbf{RQ1:} How does the smoothness of Progressive Web Apps compare to Mobile applications?

\item \textbf{RQ2:} How does the memory performance of Progressive Web Apps compare to Mobile applications?

```

```
\item \textbf{RQ3:} How does the CPU usage of Progressive Web Apps compare to Mobile applications?
\end{itemize}

As Chrome browser holds more than 60\% of the browsers market share on mobile, the performance of Progressive Webb Apps will be evaluated when running on Chrome browser in this thesis.

\paragraph{}
The global challenge behind these research questions is to collect relevant performance metrics. For this, we need different versions of an application running in a controlled environment, and tools that monitor the performance metrics.
\paragraph{}
We address the first challenge with 3 applications: a Native Android app, a cross-platform app developed with React Native and a Progressive Web App developed with ReactJS.
To address the second challenge, we investigate several techniques and tools. Many tools already exist to monitor the performance of mobile applications on Android. The challenge resides in monitoring the performance of Progressive Web Applications. For the memory performance, we inspect the usage of tools specific to mobile applications on Progressive Web Apps. For the CPU usage, we examined tools specific to web applications and compare them to mobile applications tools. For the smoothness performance, there were no frameworks or tools able to assess the smoothness of Progressive Web Apps at the time of writing.
\paragraph{}
The key technical contribution of this thesis consists in the elaboration of a model of Chrome frame rendering workflow and a tool \textit{FrameTracker} which follows the frames and extract key timestamps. They can be used by other developers to get detailed information of their web applications rendering process, and by other researchers
to understand more deeply the rendering process of web applications and compare it to mobile applications.

%To this end, we build an empirical model of the frame rendering process of the browser, that is the workflow by which a frame goes through until it is displayed. This model process events recorded by Chrome Tracing tool and extracts key timestamps of the frames rendered. It can be used by other developers to get detailed information of their web applications rendering process and identify bottlenecks \todo[inline]{the botleneck example is not clear. It seems that your model also adds more information regarding to the frames beyond the timestamps, only with the timestamps I think you cannot identify bottlenecks. And BTW, is it possible to speak about extending the information from the model beyond timestamps in the Future work?}, and by other researchers
%to understand more deeply the rendering process of web applications and compare it to mobile applications.



%\paragraph{}
%Many challenges lies behind these research questions. Since no documentation was found on the rendering pipeline of Chrome, the first step toward answering \textbf{RQ1} is to build a model empirically. Only then we can find a smoothness metric able to compare the smoothness of Mobile and Progressive Web Apps. Another challenge is to find appropriate tools to measure accurately the memory and CPU usage of Progressive Web Apps.

%\paragraph{}
%The first part of this research was to find a metric able to compare the smoothness of Mobile and Progressive Web Apps. To this end, we built a model of Chrome's rendering pipeline and deduced a smoothness metric from it. Then, the smoothness, memory performance and CPU usage was evaluated during different scenarios on 3 different applications: a Progressive Web App, a Cross-platform app developed with React Native and a Native app.

\paragraph{}
The novelty of this work resides in several points. At the time of writing, no known research has studied the smoothness, the memory performance nor the CPU usage of
```

```
Progressive Web Apps. Moreover, the method used to compare the smoothness of the
applications has never been used before, as far as we know. The model of Chrome's
rendering pipeline built empirically is also novel to the best of our knowledge.
125
126 \paragraph{}
127 Our experiments based on 3 applications, reveal that Progressive Web Apps are less
smooth than Native and Interpreted applications on Android. The difference is
significant and will be noticeable in applications requiring a high smoothness
performance, for example games or heavy animations. However, the difference
perceived by the end-user in simple applications remains to be studied. Meanwhile,
PWAs and Interpreted applications have similar CPU and memory consumption. We
observe a difference of 20\% in favor of Progressive Web Apps. However, the
difference reach 200\% between Native and Progressive Web Apps, making the Native
applications still the most performant in terms of resource consumption.
128
129 \paragraph{}
130 This work is structured as follows. \hyperref[ch:background]{Chapter 2} introduces
the concepts and the academic background related to this research.
\hyperref[ch:methodology]{Chapter 3} presents the methods used to overcome the
different challenges and answer the research questions. \hyperref[ch:model]
{Chapter 4} outlines the empirical model of the frame rendering workflow on Chrome.
\hyperref[ch:results]{Chapter 5} describes the results obtained during this study.
\hyperref[ch:discussion]{Chapter 6} discuss those results and
\hyperref[ch:conclusion]{Chapter 7} concludes this work.
131
132 %\paragraph{}
133 %The results show that Progressive Web Apps are less smooth than Native and Hybrid
applications. However, they consume similar, or smaller depending on the scenario,
amount of CPU and memory than Hybrid applications. Nonetheless, the performance of
Progressive Web Apps do not compare to Android Native applications in every aspect
studied here.
134
135 \chapter{Background}
136 \label{ch:background}
137
138 The aim of this chapter is to introduce the terminology and concepts used in this
study. We will also present the work previously done in the field of Progressive Web
Apps and analyze the tools available to measure the performance of Android mobile
applications and PWA.
139
140 \section{Mobile applications}
141 \subsection{Development}
142 There are a number of ways to develop a mobile application, the most common one
being to develop it natively, i.e. develop the mobile application once for each
targeted platform (iOS, Android or Windows Phone) using their respective
environment, Software Development Kit and programming languages. Since this can be
quite costly, an alternative has emerged in the form of cross-platform development,
that is using the same code to build the application for several platforms.
143
144 Traditionally, mobile cross-platform development is divided into four different
approaches \cite{CrossPlatform_dev}:
145 \begin{itemize}
146 \item Web: The web app is run by the browser on the mobile device. PWAs are an
improvement of this approach.
147 \item Hybrid: The application is built using web technology but executed inside a
native container. The app is rendered through full screen web-views. Frameworks like
PhoneGap and Ionic \cite{crossplatform_approaches} use this approach.
148 \item Interpreted: The application code is run by an interpreter which uses native
components to create a native user interface. Frameworks like React Native,
NativeScript and Titanium \cite{emulating_native_w_crossplatform} use this
approach.
149 \item Cross-Compiled: The source code is compiled into a native application by a
cross-platform compiler. Xamarin \cite{crossplatform_approaches} uses this approach.
150 \end{itemize}
151
152
```

```
153 \subsection{Emulators}
154 It is considered a largely accepted best practice to test an application under
    development before production. This can be done using physical devices or emulators.
    An emulator is a software that simulates the OS and the hardware capabilities of a
    devic e\cite{emulator_def}. This can be a great way to automate testing among a
    large range of smartphones. Most of the Android emulators found during this research
    are targeted at gamers and enhance the hardware capabilities usually found in the
    simulated physical device. Only a few can be used by developers to test the
    performance of their applications among different smartphones:
155
156 \paragraph{}
157 \textbf{Android Emulator}\footnote{Android Emulator:
    https://developer.android.com/studio/run/emulator/} is the official emulator for
    Android provided by Google along Android Studio. It offers the latest Android APIs
    but a limited range of physical devices. However, it is possible to customize
    hardware characteristics and the device's look.
158
159 \paragraph{}
160 \textbf{Genymotion}\footnote{Genymotion: https://www.genymotion.com/desktop/}
    contains a wider range of smartphones but fewer APIs than Android Emulator.
    Genymotion can be linked with Android Studio in order to test in-development apps.
    It is also possible to customize hardware characteristics.
161
162 \paragraph{}
163 \textbf{Visual Studio Android Emulator}\footnote{Vsial Studio Emulator:
    https://docs.microsoft.com/fr-fr/visualstudio/cross-platform/visual-studio-emulator-
    for-android?view=vs-2015} is the solution offered by Microsoft on Visual Studio.
    However, it is not supported since Visual Studio 2017. They recommend using the
    official Android Emulator instead. It offers a limited set of hardware and API
    configurations that matches several smartphones at once.
164
165
166 \section{Progressive Web Apps}
167
168 \subsection{Concept}
169
170 A Progressive Web App, or PWA for short, is a regular web application with a few
    more features such as offline and install capacity. Its aim is to reduce the gap
    between mobile and web development. Once it is installed, a PWA should behave just
    like a native app for the end-users. Thus, the end-user should not be aware of the
    browser running the app behind the scenes, and be able to launch the app without
    internet connectivity. \newline
171 The concept of Progressive Web Apps holds a lot of potential
    \cite{PWApossibleUnifer}. It could allow developers to use the same code for the
    web, the mobile, and the desktop app depending on browser implementations. This
    would considerably reduce the development and maintenance cost of a single
    application. Moreover, deployments and updates would not have to go through the app
    stores to be available to users, further reducing the overall cost of the
    application and increasing its access.
172 \paragraph{}
173 In order to be called Progressive, a web application has to implement several
    feature s\cite{PWA_def}:
174
175 \begin{itemize}
176 \item \textbf{Progressive:} Work for every browser
177 \item \textbf{Responsive:} Fit any screen size
178 \item \textbf{Connectivity independent:} Be able to work offline or with low-
    connectivity thanks to a service worker
179 \item \textbf{App-like:} Have a Native-like interaction by using the app-shell model
180 \item \textbf{Fresh:} Keep the app up-to-date with the service worker
181 \item \textbf{Safe:} Be served over TLS to prevent snooping and content tampering
182 \item \textbf{Discoverable:} Be identifiable as ''applications'' thanks to W3C
    manifests and service workers
183 \item \textbf{Re-engageable:} Use re-engagement features like push-notifications
184
```

```
\item \textbf{Installable:} Save the app on the home screen without going through an app store
\item \textbf{Linkable:} Share the app with a simple URL
\end{itemize}

\subsection{Architecture}

In practice, Progressive Web Apps contains 3 main components: the web app manifest, the service worker and the app shell.

\medskip
\textbf{App Manifest} \newline
The web app manifest is a JSON file containing the metadata concerning the native display of the app once installed on the user's phone. In this file, the developer can define the app's metadata such as the splash screen, the app icon, the theme color and the app title. Without it, the app is uninstallable.

\medskip
\textbf{Service Worker} \newline
The service worker is the central part of a PWA. It is responsible for the new background features such as push notifications, offline experience and background synchronization\cite{SW_def}. The service worker acts as a proxy server for the PWA: it intercepts requests, caches the response or gives the already cached response. As a worker, it does not have access to the DOM and works on a different thread as the main app. This is especially useful not only for offline capability but for background optimization tasks.

\medskip
\textbf{App shell} \newline
The app shell is essentially the User Interface of the app without any content \cite{AppShell_def} (the data and images fetched remotely). By caching it via the service worker, it offers the user a better performance with instant loading of the UI on repeated visits and a better low-connectivity experience.

%\todo[inline]{Change name, "State of the art", "Related work"}

\section{Profiling tools}

Profiling tools are software or command-line tools that help developers evaluate the performance of their app and identify performance bottlenecks. They measure metrics such as the CPU, the memory used or functions called during a recording.

\subsection{Android}

Few tools are available to profile Native and cross-platform applications on Android aside from Android Profiler \cite{nanoscope}. This software can provide real-time information about CPU, Memory, Network and Battery consumption in the form of graphs. While it does not require any specific install as it comes with Android Studio and provides a User Interface, it can have a high overhead \cite{nanoscope} and make the app less performant during recording than it is usually.

\paragraph{}
Android also provides other command-line tools to profile applications. Most of them are only available through Android Debug Bridge (ADB)\footnote{ADB: https://developer.android.com/studio/command-line/adb} which allows a computer to communicate with a device either to get information or execute commands.

\paragraph{}
For example, dumpsys\footnote{Dumpsys: https://developer.android.com/studio/command-line/dumpsys} can provide a lot of information about the device or the processes currently running. This information is available through the system services such as \textit{meminfo} (memory), \textit{cpuinfo} (CPU usage), \textit{gfxinfo} (animation), \textit{netstats} (network) or \textit{batterystats} (battery).
```

```
219
220 \paragraph{}
221 Another example is top. It comes from the Linux command-line of the same
    name\footnote{Ubuntu manuals:
    http://manpages.ubuntu.com/manpages/xenial/man1/top.1.html}, but with limited
    options. It displays the process activity in real-time such as CPU, memory and
    process priority.
222
223 \paragraph{}
224 Systrace\footnote{Systrace:
    https://developer.android.com/topic/performance/tracing} is also a useful tool and
    does not require ADB. It records a device activity and outputs an HTML file which
    can be viewed on Chrome browser. This file displays a timeline of events for each
    thread running during the recording, the CPU activity divided for each CPU, frames
    and other events depending on the categories of events selected. It is mainly used
    to identify the cause of bottlenecks in an application.
225
226 \paragraph{}
227 While not a profiling tool in itself, Monkeyrunner\footnote{Monkeyrunner:
    https://developer.android.com/studio/test/monkeyrunner} is a useful tool when
    testing an application. Its purpose is to automate testing by interacting with the
    device from a Python script. It can also take snapshots or execute \textit{adb
    shell} commands, such as \textit{dumpsys} and top viewed previously.
228
229 \subsection{Progressive Web App}
230
231 Even though Progressive Web Apps can be installed and managed like a native
    application for the end-user, they are still Web applications. Thus, they have their
    own set of profiling tools provided by the browser. Since this study is limited to
    PWAs running on Chrome, this section will focus on Chrome's profiling tools. They
    may not work on other browsers.
232
233 \paragraph{}
234 One of them is the Chrome's DevTools, a complete set of developer tools available
    on Chrome Browser. It is divided into several panels, each with its own purposes.
    For example, the \textit{Network} panel displays all requests done by the web app,
    the \textit{Console} panel displays debug and error messages and the
    \textit{Performance} panel \cite{chrome_devtools_perf} displays an overview of the
    app performance, such as a timeline of function calls, a CPU graph, a memory graph
    and information regarding frames.
235
236 \paragraph{}
237 Lighthouse \cite{lighthouse} is another tool of the Chrome's DevTools, available in
    the \textit{Audit} panel. Though its purpose is to automatically assess any web
    page's quality (Best Practices, Performance, Search Engine Optimization,
    Accessibility, Progressive Web App), its main target are Progressive Web Apps
    \cite{PWApossibleUnifer}.
238
239 \paragraph{}
240 Chrome DevTools Protocol\footnote{Chrome DevTools Protocol:
    https://chromedevtools.github.io/devtools-protocol/} is a set of commands and events
    used to communicate with the browser and the web page. Chrome DevTools takes its
    source of information from it. A number of other tools relies on it for their
    features \cite{awesome_CDP}. It may be used for browsers other than Chrome if they
    are based on Blink.
241
242 \paragraph{}
243 Telemetry\footnote{Telemetry: https://github.com/catapult-
    project/catapult/blob/master/telemetry/README.md} is a framework used for automated
    performance testing on Chrome. It can be used on several platforms (desktop or
    Android). It can automatically interact with a web page while taking measurements.
244
245 \section{Rendering Pipeline}
246 \label{def:frame}
247
```

```
248 The smoothness of an application greatly depends on the frame rate. A frame is an
    image displayed by the application on the screen. Each time something on the screen
    needs to be changed, a new frame is produced and displayed. We call that rendering.
    The faster an app can produce a new frame, the faster it can display a response to
    user input and the more animations it can properly handle without blocking user
    interaction. \newline
249 Though the goal of Progressive Web Apps is to look and behave the same way as mobile
    application once installed on a smartphone, they rely on different technologies to
    do so. Thus, the rendering pipeline (the stages by which a frame is produced) can
    differ.
250
251
252 \subsection{Android} \label{background:pipeline:android}
253 \label{def:surfaceflinger}
254 The Android Graphics pipeline is divided between 3 main components: the Application
    Renderers \badge{1}, SurfaceFlinger \badge{2} and the Hardware Composer \badge{3}
    \cite{android_graphics}.
255 When an application wants to display something on the screen, its Application
    Renderer \badge{1} (usually provided by Android framework) gives a buffer of
    graphical data to SurfaceFlinger \badge{2}. SurfaceFlinger then takes all
    available buffers and composes them into a single buffer that it passes on to the
    Hardware Composer \badge{3} (see \autoref{fig:android_data}). The Hardware Composer
    \badge{3} is responsible for actually displaying the buffer on the screen.
256 \newline
257
258 \begin{figure}[!ht]
259 \includegraphics[width=13cm]{kththesis/Figures/android_graphics_pipeline.png}
260 \caption[Graphic data flow through Android]{Graphic data flow through Android
    \footnotemark}
261 \label{fig:android_data}
262 \end{figure}
263
264 \footnotetext{Source: \url{https://source.android.com/devices/graphics}}
265
266 Everything displayed on the screen has to go through SurfaceFlinger and the
    Hardware Composer.
267 All these components synchronize with each other thanks to the VSYNC signal
    \cite{vsync}. When VSYNC signal is fired, all three components wake up at the same
    time: the Hardware Composer displays frame N, SurfaceFlinger takes a look inside
    the BufferQueue and composes frame N+1 if a new buffer is available, and the App
    Renderers generate frame N+2. The VSYNC signal depends on the device refresh rate,
    usually set at 60Hz though some new devices also support 90 or 120Hz
    \cite{refresh_rate}.
268
269 \paragraph{}
270 Application rendering itself is divided into 7 different stages
    \cite{app_rendering}:
271 \begin{enumerate}
272 \item Input handling: executes input events callbacks
273 \item Animation: executes animators callbacks
274 \item Measurement and Layout: computes the size and position of all items
275 \item Draw: generates the frame's drawing commands in the form of a display list
276 \item Sync and Upload: transfers objects from CPU memory to GPU memory
277 \item Issue commands: sends the drawing commands to the GPU
278 \item Process and Swap buffers: pushes the frame's graphical buffer into the
    BufferQueue shared with SurfaceFlinger
279 \end{enumerate}
280
281 The Application rendering of Android's framework is synchronous as each stage will
    happen one after another. Only Input handling and Animation may not always happen
    depending on inputs and callbacks.
282
283 \subsection{Chrome}
284
```

```
Before understanding Chrome's rendering pipeline, it is best to first understand its
architecture. It is based on multiple processes running at the same time, with each
its specific function \cite{chrome_architecture}
(\autoref{fig:chrome_architecture}):

\begin{itemize}
\item Browser: controls everything related to the browser such as address bar,
network requests and file access.
\item GPU: displays all the elements on the screen by calling the OS's graphics
library.
\item Renderer: runs the web application. Each tab has its own Renderer sandboxed
process with limited rights to protect the user and avoid crashing the browser.
\item Plugin: runs the plugins used by the web application.
\end{itemize}

\begin{figure}
\centering
\includegraphics[width=13cm]{kththesis/Figures/chrome_architecture.png}
\caption[Chrome's multi-process architecture]{Chrome's multi-process architecture
\footnotemark}
\label{fig:chrome_architecture}
\end{figure}

\footnotetext{Source: \textit{Inside look at modern web browser (part 1)}
\cite{chrome_architecture}}

Each process also has several threads running at the same time for optimization.

\paragraph{}
Chrome's rendering pipeline \cite{chrome_pixel} contains similar stages as Android's
application rendering but with some differences:
\begin{enumerate}
\item Parsing: translates the HTML into a DOM Tree.
\item Styling: applies CSS style to DOM elements.
\item Layout: computes the geometry of DOM elements (size and position) and produces
a layout tree.
\item Compositing: decomposes the page into layers which can be independently
painted and rastered.
\item Pre-painting: builds the property tree to apply to each layer.
\item Painting: records paint operations into a list of display items.
\item Tiling: divides the layers into tiles making its raster less expensive if only
a part of it is needed
\item Raster: executes the paint operations and decodes images to produce a bitmap.
\item Drawing: generates a Compositor Frame and sends it to the GPU process.
\end{enumerate}
Every Renderer and the Browser Process can compute frames according to the pipeline
described above. The GPU processes then aggregates all Compositor Frames according
to the surface they represent on the screen and sends calls to the OS's graphics
library. For Android, that means pushing a buffer of graphic data to
SurfaceFlinger.
\newline
As opposed to Android, the output of the different stages are reused whenever
possible. The frame is divided so that a small change in the frame only triggers a
small amount of work to render it.
For example scrolling only changes the position of a layer. There is no need to go
through Parsing, Styling and Layout again. Pre-painting, Painting, Tiling and
Raster may also be skipped if the size of the frame computed previously was bigger
than the display screen.

\section{State of the art}

This section will first present the academic research related to Progressive Web
Apps and then describe other works that evaluated the performance of cross-platform
frameworks.
```

```
325
326 \subsection{Progressive Web Apps}
327 Though Progressive Web Apps have sparked a real interest in the mobile and web
    development industry, only a few research papers studied
    them\cite{PWApossibleUnifer, Biorn-Hansen2, Biorn-Hansen3}.
328
329 \paragraph{}
330 Biørn-Hansen, Machrazk and Grønli tried to raise interest in the academic
    community with three successive papers: \textit{Progressive Web Apps: The Possible
    Web-native Unifier for Mobile Development} \cite{PWApossibleUnifer}.
    \textit{Progressive web apps for the unified development of mobile applications}
    \cite{Biorn-Hansen2} and \textit{Progressive Web Apps: the Definite Approach to
    Cross-platform development?} \cite{Biorn-Hansen3}. Their first two papers also
    included a study of PWA's performance compared to other mobile development
    approaches, namely hybrid and interpreted for their first paper, and native, hybrid,
    interpreted and cross-compiled for their second. They looked at launch time, the
    size of installation and the app-icon-to-toolbar-render time with an online
    stopwatch. While the PWA was a lot smaller and launched faster than any other app,
    the time from app icon to toolbar render depended on whether the browser was already
    running on the background. If so, the PWA was the fastest and if not, was slower
    than the native and interpreted app.
331
332 \paragraph{}
333 Kressens \cite{PWAapplicability} compared the First Paint metric\footnote{Time to
    render the first pixel} of a Progressive Web App, a regular Web App and a Native App
    on both Android and iOS with a fast and slow network, as well as the energy
    consumption of a PWA and Native applications on iOS and Android. He found that the
    PWA launched faster than the regular Web App and the Native Android app, and
    slightly slower than the Native iOS App. The energy consumption was similar between
    the PWA and both Native apps but slightly lower for the PWA than the Native Apps.
    The overall conclusion of this research is that PWAs are a viable alternative to
    native applications.
334
335 \paragraph{}
336 Yberg \cite{YbergViktor2018NPaU} compared manually the launch time of an android
    native application and a PWA with and without the browser in memory. The PWA was
    faster (with the browser in memory) or as fast as the native application. \newline
337 Malavolta, after presenting Progressive Web Apps as a mobile development strategy
    \cite{malavolta2016beyond}, focused on the impact of service workers on PWA's
    energy consumption and found no significant impact\cite{SW_and_energy}. \newline
338 Fransson and Driaguine \cite{PWAbc_responsetime} compared the response time of
    PWAs and Native Android Applications when accessing the camera and geolocation.
    The Progressive Webb App was faster at using geolocation than its native
    counterpart. On the contrary, camera access was faster on the native app than the
    PWA. \newline
339 Johannsen \cite{JohannsenFabian2018PWAa} evaluated the code complexity brought by
    Progressive Web Applications to a regular application using different metrics such
    as the cyclomatic complexity and Halstead effort. He concluded that the added
    complexity was low. \newline
340 Lee et al. \cite{Pride_Prejudice} explored the security system behind the push
    notification system for Progressive Web Apps and found several concerning flaws
    which they reported to the vendors. \newline
341 Gambhir and Raj \cite{gambhir2018analysis} analysed the impact of service workers
    on the performance of Progressive Web Apps and compared it to Android applications,
    especially the response time of the servers. They found that PWAs could perform
    better than Android applications. \newline
342 Lastly, at the time of writing only two research papers examined the user experience
    offered by Progressive Web Apps. Cardieri and Zaina \cite{PWA_UX_comparison_study}
    conducted a qualitative analysis of user experience on three platforms: web mobile,
    native android and PWA. They concluded that there was no significant difference of
    user experience between the platforms. Fredrikson
    \cite{emulating_native_w_crossplatform} also supports this conclusion with a
    quantitative and a qualitative study comparing user experience with a React Native
    App, a Native Android App and a Progressive Web App.
343
344 \subsection{Performance of Cross-platform applications}
345
```

```
346 The performance parameters used to compare mobile applications can differ greatly
    between studies.
347
348 \paragraph{}
349 Biørn-Hansen, Grønli and Ghinea \cite{animation_performance} analyzed the metrics
    commonly used in animation performance evaluation: frames per second (FPS), CPU
    usage, device memory usage and GPU memory usage. They also used those metrics to
    evaluate the animation performance of several mobile applications developed in
    Native Android, Native iOS, React Native, Ionic and Xamarin. They concluded that FPS
    and GPU memory were not really useful to compare their performance, as the GPU
    memory did not change much between the animations and the FPS was not relevant to
    short-running animations like the ones they tested. The results suggest that React
    Native is the most performant cross-platform framework on Android while it is Ionic
    on iOS.
350 %However, they did not compare the performance of Native and Cross-platform
    frameworks.
351 \paragraph{}
352 Willocx et al. \cite{willocx2016comparing} evaluated the performance of 10
    different cross-platform frameworks (divided between Javascript frameworks and
    source code translators) against native on iOS, Android and Windows Phone. The
    performance parameters used were the response time, the CPU usage, the disk space
    and the battery usage. Their reached several conclusions from their experiments.
    First, Javascript frameworks consumes the most CPU and memory, and launch the
    slowest. However, their response times are similar to native applications during
    run-time. Second, all Cross-platform frameworks use more persistent memory than the
    native applications. Third, the performance of cross-platform frameworks depends on
    the targeted platform and version. Lastly, they conclude that the performance
    overhead of cross-platform frameworks is acceptable, especially on high-end devices.
353
354 \paragraph{}
355 Ciman and Gaggi \cite{ciman2017empirical} compared the energy consumption of 5
    applications on iOS and Android: a web Application, a Hybrid app using PhoneGap, an
    Interpreted app using Titanium, a Cross-compiled app using MooSync and a Native
    app. Their results show that cross-platform applications always consume more energy
    than a native application. Moreover, the update of the User Interface consumes more
    energy than retrieving data from sensors.
356
357 \paragraph{}
358 Delia et al. \cite{delia2017approaches} evaluated the performance of Cross-platform
    frameworks with their processing speed. They tested 7 different frameworks (Native,
    Web, Cordova, Titanium, NativeScript, Xamarin, Corona) on iOS and Android. The
    results showed a great difference between iOS and Android. On iOS, the Native
    application was by far the most performant, followed by Corona, Web or Xamarin
    (order depending on device). On the contrary, Xamarin, Corona and Native were the
    less performant on Android. Across all devices and OS tested, the Web app displayed
    a good performance.
359
360
361 \section{Problem Analysis}
362
363 Though some security issues need to be resolved \cite{Pride Prejudice}. Progressive
    Web Apps have the potential to become a popular cross-platform solution for
    developing mobile applications. They do not add much complexity to a regular web app
    \cite{JohannsenFabian2018PWAa}, greatly reduce the cost of developing a mobile
    application for every platform, and have similar performance than mobile
    applications regarding launch time \cite{PWApossibleUnifer}\cite{Biorn-Hansen2}
    \cite{PWAapplicability}, energy consumption \cite{PWAapplicability}, user
    experience \cite{emulating_native_w_crossplatform}\cite{PWA_UX_comparison_study},
    or even hardware access in some cases \cite{PWAbc_responsetime}.
364 \paragraph{}
365
366 However, at the time of writing no study that examined the performance at run-time
    was found (except for energy consumption). This project will compare Android Native,
    Interpreted and Progressive Web Apps thanks to several performance parameters,
    namely the smoothness (how fast the application can render new frames), the memory
    performance and the CPU usage.
367
```

```
368 The smoothness of a video is usually evaluated using the FPS rate, or frames per
    second \cite{smooth_gui}. It is also one of the main metrics recommended by
    Android\footnote{https://developer.android.com/training/testing/performance} and
    Google\footnote{https://developers.google.com/web/tools/chrome-devtools/evaluate-
    performance} to analyze the run-time performance of an application.\newline
369 However, this metric is not always considered relevant. Biørn-Hansen et al.
    \cite{animation_performance} examined several metrics used to evaluate animation
    performance in mobile application. They concluded that FPS was not a really useful
    information, especially for animations that runs only shortly and leave the
    application idle the rest of the time. Lin et al.\cite{smooth_gui} preferred to
370 analyze the smoothness performance with jank, i.e. the number of non-updated
    screens displayed. Wen \cite{smoothnessQoE} instead considered 6 different indexes
    related to the frame intervals instead of the frame time.
371
372 As there is no real consensus on the metrics to be used to analyze smoothness
    performance, the first part of this study will focus on finding a relevant and
    measurable metric to compare the smoothness of Android applications and Progressive
    Web Apps.
373
374 \paragraph{}
375 This chapter provided the necessary background to conduct this research and
    additional motivation. The next chapter will focus on the method of comparison of
    PWAs and Mobile Applications and the tools used to do so.
376
377
378 \chapter{Methodology}
379 \label{ch:methodology}
380
381 The aim of this study is to compare the smoothness of Mobile and Progressive Web
    Applications, as well as the memory performance and the CPU usage. First, we will
    define the metrics used to evaluate the smoothness performance, the memory
    performance and the CPU usage. Then, we will present the protocols used to collect
    those metrics. Finally, we will describe the benchmark applications and experiments
    used to answer the research questions.
382
383 %The main challenge behind this goal is to define a metric able to compare the
    smoothness of Mobile and Progressive Web App. For this, a model of the Chrome's
    rendering pipeline had to be built. This chapter will outline the reasons behind the
    necessity of this model and the method used to built it. It will also describe the
    methods and experiments used to evaluate the CPU usage, the memory usage and the
    smoothness of Mobile and Progressive Apps.
384
385 \section{Metrics}
386
387 To answer the research questions outlined in the Introduction, we need to identify
    relevant metrics able to evaluate the smoothness performance, the memory performance
    and the CPU usage. This section will describe the different metrics chosen to
    evaluate them and the reasons behind this choice.
388
389 \subsection{Smoothness performance}
390
391 The smoothness of an application according to Android\cite{app_smoothness} is
    closely related to the \hyperref[def:frame]{frames} displayed by the application.
    If a frame takes too long to be rendered, the user might notice some stuttering
    where motions are visibly fragmented or freezing where the application halts for a
    long time, resulting in a poorer quality of the User Experience.
392
393 \paragraph{}
394 The smoothness of web or mobile applications are often evaluated using the FPS rate
    that is the number of frames per second. However, as Biorn-Hansen et al.
    suggested\cite{animation_performance}, simply looking at the FPS rate is irrelevant
    of the smoothness of applications as the animations are sparse and run shortly.
395
396 \paragraph{}
397
```

```
Thus, to evaluate the smoothness performance, we use the frame rendering duration,
that is the amount of time the applications start computing a new frame until the
frame graphic buffer is pushed to \hyperref[def:surfaceflinger]{SurfaceFlinger}.
The frame rendering duration will be computed from the average frame rendering
duration extracted from 100 experiments that we will describe in
\autoref{method:benchmark}.
398
399 %, a component of Android Graphics which handles what is displayed on the screen.
400
401 %\paragraph{}
402 %
403
404
405 %\todo{Here you are defining the metrics used, so this sentences can be the end of
    the collection subsection. The same for the other RQ}
406 %\todo[inline, color=cyan!20]{It was the description you put for the research
    questions when you described the new layout but okay}
407
408 \subsection{Memory performance}
409
410 The memory performance of an application is defined by the amount of RAM it consumes
    when it runs. As this resource is limited on smartphones and a high RAM consumption
    can increase the energy consumption, mobile applications need to use it efficiently.
411
412 %\paragraph{}
413 Thus, we will evaluate the memory performance of the applications by the total
    amount of RAM used by the applications, computed from the average of a 100 metric
    collection.
414
415 \subsection{CPU usage}
416
417 A high CPU usage in mobile phones often results in high energy consumption, a very
    limited resource in smartphones. Thus, it is important that mobile applications use
    the CPU efficiently. CPU usage can be evaluated with 2 metrics: the percentage of
    total CPU consumption, or the percentage of CPU core consumption. They are linked
    with the equation:
418
419 \[\% \ total\ CPU= \frac{\% \ CPU\ core}{number\ of \ CPU \ cores} \]
420
421 As the percentage of total CPU consumption depends on both the number of CPU cores
    and the CPU cores capacity, and the percentage of CPU core consumption depends only
    on the capacity of the CPU cores, the latter will be used to evaluate the CPU usage
    of the applications. It will allow a more relevant comparison between devices.
422
423 \paragraph{}
424 Thus, we will evaluate the CPU performance of the applications by the percentage of
    CPU core used by the applications, computed from the average of a 100 metric
    collection.
425
426
427
428 \section{Metric collection}
429
430 In the previous section, we defined relevant metrics able to evaluate the
    performance of Mobile and Progressive Web Applications. However, it is also
    necessary to collect them with enough accuracy. This section will discuss the
    methods used to extract those metrics on Native, Interpreted and Progressive Web
    Apps.
431 \subsection{Frame duration}
432 \label{method:smoothness}
433 %\todo[inline]{Android (Explain the Android rendering pipeline as it is written in
    the chapter 3 now. Then how to collect the frame duration)PWA What is the model ?
    How to build the model, etc? How to collect the Answer how to compare data given the
```

```
metric, for example, it can be using a binary operation, "if data from A is greater than data from B then A is better than B"}

\subsubsection{Mobile} \label{collection:android}
The main source of graphical information for mobile applications is the service \textit{gfxinfo}. This service outputs several statistics about the application's frames, such as total number of frames rendered, the percentage of janky frames\footnote{Frames that were dropped or delayed} or the different timestamps of the most recent frames. Those timestamps can help developers identify the most time-consuming stage of the rendering pipeline and thus improve the smoothness of their application. \newline
\indent Another method to follow the computation of a frame on Android is to use the tool Systrace. With the relevant options selected, a developer can visualize the different functions called to compute a frame in a timeline as a flamegraph (see \autoref{fig:android_systrace_zoom}).

\begin{figure}
\centering
\includegraphics[width=13cm]{kththesis/Figures/android_systrace_zoom.JPG}
\caption{Android Systrace - Screenshot}
\label{fig:android_systrace_zoom}
\end{figure}

\begin{figure}[!ht]
\centering
\includegraphics[width=13cm]{kththesis/Figures/Android_frame_timeline.png}
\caption{Linking Systrace and frame's timestamps - Screenshot with timestamps added}
\label{fig:android_frame_timeline}
\end{figure}

\paragraph{}
By looking at Android source code\footnote{\url{https://github.com/aosp-mirror/platform_frameworks_base/blob/master/core/java/android/view/Choreographer.java}}, it is possible to link the timestamps gathered by \textit{dumpsys gfxinfo} to the events displayed by Systrace (see \autoref{fig:android_frame_timeline}), giving an overview of the path taken by a frame on Android. \newline
\indent Every frame starts with a VSYNC signal (first vertical red line in \autoref{fig:android_frame_timeline}), which acts as was presented in \autoref{background:pipeline:android} as a wake-up call for all the components of Android Graphics. The UI thread handles input and animation before measuring and computing the layout of the graphical elements. It updates the frame and hands it to the Renderer thread which issues the drawing commands and pushes the graphic buffer to SurfaceFlinger. Then, SurfaceFlinger is responsible for displaying the new frame on the screen and the application has no control over it. To sum up, for a native and hybrid application on Android a frame starts at VSYNC signal and is completed when its graphic buffer is pushed to SurfaceFlinger.

\paragraph{}
The start of the frame rendering time differs depending on what one thinks of as the start of the frame. The VSYNC signal is the moment the application decides to render a new frame. The timestamp \textit{HandleInputStart} marks the start of deciding what will change from the previous frame. The timestamp \textit{PerformTraversalStart} indicates the beginning of the computation of the frame. \newline

%\todo[inline, color=cyan!20]{Changed the conclusion. What about now? Better}
In the context of comparing the smoothness of Mobile Applications and Progressive Web Apps, the start of the frame can only be identified with the frame rendering process of both Android and Chrome. Thus, the start of the frame on Android can be determined only after inferring the frame rendering process of Chrome.

\subsubsection{Chrome}

\paragraph{}
```

```
466 We can see on \autoref{fig:pwa_systrace_zoom} that the frames on Chrome follow a
    different pipeline and thus remain undetected by Android system. Nonetheless, we can
    observe common functions in the Systrace flamegraphs of Android applications and
    Chrome. Both of them use the same function to push a frame graphic buffer to
    SurfaceFlinger, displaying a new frame. Thus, if we gather similar timestamps for
    Chrome's frames than for Android's, especially the start and end of the frame, it
    would be possible to compare the smoothness of the applications.
467
468 \begin{figure}[h]
469 \centering
470 \includegraphics[width=13cm]{kththesis/Figures/pwa_systrace_zoom.JPG}
471 \caption{Chrome Systrace for a PWA application execution - Screenshot}
472 \label{fig:pwa_systrace_zoom}
473 \end{figure}
474
475 \paragraph{}
476 However, at the time of this writing, there was no documentation nor tool that
    provided a similar overview of the workflow of the generation of a frame on Chrome.
    Thus, the first step toward comparing the smoothness of Mobile and Progressive Web
    Apps is to build a model of this workflow.
477
478 \paragraph{}
479 This first model of Chrome frame rendering workflow was built using Chrome Tracing
    tool, which records Chrome's processes activity in the form of events with a name, a
    timestamp, a duration and arguments. By analyzing those events, especially their
    names and arguments, we deduced a first model of Chrome frame rendering workflow.
    However, this model was deduced only from a few frames. A tool was needed to test it
    on a larger scale.
480
481 \paragraph{}
482 This tool, \textit{FrameTracker}, was developed for two main goals: check the
    consistency of the model with the events observed, and collect the frames rendering
    duration. It takes as input a list of events recorded by Chrome Tracing tool during
    a small period of time and outputs a list of timestamps representing the frames
    computed during the recording. At the same time, it will check that the events match
    the identified workflow, thus confirming or invalidating the current model.
483
484 \begin{figure}[h!]
485 \centering
486 \includegraphics[width=8cm]{kththesis/Figures/chrome_model_example.png}
487 \caption{Example of workflow}
488 \label{fig:model_example}
489 \end{figure}
490
491 \begin{figure}[h!]
492 \centering
493 \includegraphics[width=13cm]{kththesis/Figures/frametracker_example.png}
494 \caption{example of FrameTracker}
495 \label{fig:frametracker_example}
496 \end{figure}
497
498
499 \paragraph{}
500 For example, if the workflow consists of 3 events: \textit{Event1}, \textit{Event2},
    \textit{Event3} (see \autoref{fig:model_example}). \textit{FrameTracker} will look
    for those events in the list provided and build the list of frames with the
    timestamps \textit{t1}, \textit{t2} and \textit{t3} corresponding respectively to
    the occurrence of \textit{Event1}, \textit{Event2} and \textit{Event3} (see
    \autoref{fig:frametracker_example}). It will also verify that Event1, Event2 and
    Event3 happen for every frame, and at the same order.
501
502 \paragraph{}
503 The model is the identified workflow of the generation of a frame on Chrome.
    \textit{FrameTracker} is the tool used to extract the frame timestamps according to
```

```
the model and at the same time, check the correspondence of the model to the events
observed.

\begin{figure}[h]
\centering
\includegraphics[width=13cm]{kththesis/Figures/building_model_diagram.png}
\caption{Building the model and FrameTracker tool}
\label{fig:building_process}
\end{figure}

\paragraph{}
The model and \textit{FrameTracker} were developed with an iterative process (see
\autoref{fig:building_process}). An error from \textit{FrameTracker} could mean
two things: the model was not accurate and needed to be modified, or there was a bug
inside \textit{FrameTracker} that needed to be fixed. The model and/or
\textit{FrameTracker} were modified and tested with the same recording until no
errors appeared. Then, they were confronted to another recording of events.



\paragraph{}
The complete Script of FrameTracker is available on
Github\footnote{PWA\ Master\ Thesis repository:
\url{https://github.com/camilleFournier/PWA_Master_Thesis}}.
The model and \textit{FrameTracker} was first built with a single benchmark
application with limited interactions. Then, they were confronted with 10 event
recordings of 8 different PWAs from the Github repositories
pwarocks\footnote{\url{https://github.com/pwarocks/pwa.rocks}} and awesome-
pwa\footnote{\url{https://github.com/hemanth/awesome-pwa}} to include a wider range
of interactions. This confrontation was done 3 times, each with new PWAs until no
significant errors remained.

Those errors were:
\begin{itemize}
\item \textbf{Edge effects}: the start and end of a recording can happen anywhere on
a frame's timeline. Some events, which require previous events or child events
according to the model can raise an error at the start or the end of recording.
Those errors are kept to a minimum, but can still happen as the recording does not
always start or end at the same time for all threads. They do not invalidate the
model.
\item \textbf{Additional surface}: as it will be explained in more details in
\autoref{ch:model}, every frame rendered is actually composed of aggregated
surfaces. Each iframe or video embedded in the PWA adds a surface and make it more
difficult to follow the frames. As this project is limited to simple apps with no
videos or third-party advertisement, those errors are ignored. They do not
invalidate the model.
\item \textbf{Bugs}: those errors are not explained by the model. However, they are
varied and extremely scarce (1 error over 5 000 frames computed) and are not
considered statistically significant.
\end{itemize}
\paragraph{}
With this model, which we will discuss more deeply in the next chapter, we were able
to compare the smoothness of Mobile and Progressive Web Apps. To do so, we use the
frame rendering duration, defined as the time span from the start of the computation
of the frame until it is passed on to SurfaceFlinger to be displayed on the screen.
It will be computed from the average frame rendering duration extracted from 100
experiments that we will describe in \autoref{method:benchmark}. The application
with the smallest frame rendering duration will be considered the smoothest.

\iffalse
\subsubsection{Chrome's frame rendering workflow in Android}

This model was first built using Chrome Tracing tool which records the activity of
Chrome's processes during a small period of time. By analyzing the events recorded
(names, arguments, description), we build a first empirical model of Chrome's frame
```

rendering workflow on Android, that is the sequence of events leading to rendering a new frame, from the moment Chrome decides to render a new frame to the moment it pushes a graphic buffer to SurfaceFlinger.

\paragraph{}
This first model however, was constructed manually with only a few frames. A tool was needed to verify this empirical model at a larger scale, and at the same time extract the frames key timestamps in order to compute the frame rendering duration.

\paragraph{}
This tool, called \textit{FrameTracker}, takes as input the list of events recorded by Chrome Tracing tool and outputs the key timestamps of the frames rendered during the recording, as defined by the model. At the same time, it checks that the events observed match the model.

\paragraph{}
\autoref{fig:model_example} and \autoref{fig:frametracker_example} provides an example. If the model, that is the identified workflow of a frame, is composed of 3 type of events: Event1, Event2 and Event3 (see \autoref{fig:model_example}, \textit{FrameTracker} (see \autoref{fig:frametracker_example}) will look for those types of events in the list provided. It will extract their timestamps, regroup them by frames, and output those timestamps. At the same time, it will verify the model and check everything is as expected. In the example, when FrameTracker starts processing event2, it will check that there is a frame waiting for

%\textit{FrameTracker} is the tool used to extract the frame timestamps according to the model and at the same time, check the correspondence of the model to the events observed.

The model is the identified workflow of the computation of the frame on Chrome. In the example provided in \autoref{fig:frametracker_example} \todo{Add reference to figure}, it verifies that Event1, Event2 and Event3 happen for every frame, and at the same order.
With this model of the workflow of frame, which we will discuss in deep in the next chapter, we were able to define a metric able to compare the smoothness of Native \todo{I changed Mobile by Native. Check this along the document also} and Progressive Web Apps on Android.

\todo[inline]{Explain the figure}

\todo[inline]{To be sound with the results, we need to define how you detect the compoments refered in the results chapter, like "Component", CrBrowserMain, "VizCompositor", etc. What are these names? Explain this process here, something like "we conducted a manual work analysing the event names, etc, etc"}

\subsubsection{FrameTracker}

\paragraph{}
Once the model was inferred and verified, we built the tool, named \textit{FrameTracker}. It is able to extract the frame's timestamps on Chrome. The goal of this tool is to take as input a list of events recorded by Chrome Tracing tool during a small period of time and to output a list of frames \todo{I think is better to say a list of information regarding to the frames, aka the timestamps} computed during the recording with their timestamps.

For example, if the frame's workflow consists of 3 events: \textit{Event1}, \textit{Event2}, \textit{Event3} (see \autoref{fig:model_example}). \textit{FrameTracker} will look for those events in the list provided and build the list of frames with the timestamps \textit{t1}, \textit{t2} and \textit{t3} corresponding respectively to the occurrence of \textit{Event1}, \textit{Event2} and \textit{Event3} (see \autoref{fig:frametracker_example}).

```
562 \begin{figure}
563 \centering
564 \includegraphics[width=8cm]{kththesis/Figures/chrome_model_example.png}
565 \caption{Example of workflow}
566 \label{fig:model_example}
567 \end{figure}
568
569 \begin{figure}
570 \centering
571 \includegraphics[width=13cm]{kththesis/Figures/frametracker_example.png}
572 \caption{FrameTracker's workflow}
573 \label{fig:frametracker_example}
574 \end{figure}
575
576
577 \paragraph{}
578 The complete Script of FrameTracker is available on
    Github\footnote{PWA\ Master\ Thesis repository:
    \url{https://github.com/camilleFournier/PWA_Master_Thesis}}.
579 The model and \textit{FrameTracker} was first built with a single benchmark
    application with limited interactions. Then, they were confronted with 10 traces of
    8 different PWAs from the Github repositories
    pwarocks\footnote{\url{https://github.com/pwarocks/pwa.rocks}} and awesome-
    pwa\footnote{\url{https://github.com/hemanth/awesome-pwa}} to include a wider range
    of interactions. This confrontation was done 3 times each, with new PWAs until no
    significant errors remained.
580
581 Those errors were:
582 \begin{itemize}
583 \item \textbf{Edge effects}: the start and end of a recording can happen anywhere on
    a frame's timeline. Some events, which require previous events or child events
    according to the model can raise an error at the start or the end of recording.
    Those errors are kept to a minimum, but can still happen as the recording does not
    always start or end at the same time for all threads. They do not invalidate the
    model.
584 \item \textbf{Additional surface}: as it will be explained in more details later,
    every frame rendered is actually composed of aggregated surfaces. Each iframe or
    video embedded in the PWA adds a surface and make it more difficult to follow the
    frames. As this project is limited to simple apps with no videos or third-party
    advertisement, those errors are ignored. They do not invalidate the model.
585 \item \textbf{Bugs}: those errors are not explained by the model. However, they are
    varied and extremely scarce (1 error over 5 000 frames computed) and are not
    considered statistically significant.
586 \end{itemize}
587
588
589
590 \todo[inline, color=green]{separate the description of the rules you use to
    monitor and analyze the events from the fact that these rules are implemented as a
    script in a specific languages environment (i.e., separate the 'what' from the
    'how')
591 Concretely: this paragraph should discuss an algorithm and at the end you can
    mention that you implemented the algorithm in language YY, because of XXX and this
    script is publicly available.
592 For figure 3.4: I suggest to turn it into an Algorithm (in latex)}
593
594
595
596
597 \paragraph{}
598 Building the model and the tool was an iterative process (see
    \autoref{fig:building_process}). An error from \textit{FrameTracker} could mean
    two things: the model was not accurate and needed to be modified, or there was a bug
    inside \textit{FrameTracker} that needed to be fixed. The model and/or
```

\textit{FrameTracker} were modified and tested with the same recording until no errors appeared. Then, they were confronted to another recording of events.

After the model and the FrameTracker were built, we are able of collecting the frame rendering duration for PWA. This metric is defined as the time span from the start of the computation of the frame until it is passed on to SurfaceFlinger to be displayed on the screen \todo{TBD}. A smaller frame rendering duration means more time to handle input events and compute background tasks. Thus, the application with the smallest frame rendering duration will be considered the smoothest.
\fi

\subsection{Memory data}

\paragraph{}
The memory in mobile applications on Android is inspected with the service \textit{meminfo} of \textit{adb shell dumpsys}. Filtering with the application package name, it takes a snapshot of its memory and outputs detailed information about the memory used though we will only monitor the total amount of RAM used by the application.

\paragraph{}
Without filtering, this command line outputs the total amount of RAM used for each process. The processes are also ordered by section: persistent, foreground, visible and cached among others. \newline
The foreground section is the most useful. When a mobile application is running, only its process appears on the foreground. In the case of PWA, 3 processes appear on the foreground: the chrome application package, a sandboxed sub-process of chrome, and a privileged sub-process of chrome. Those represent respectively, the Browser process, the Renderer process and the GPU process. This means all three processes needs to be considered when measuring the memory used by a PWA.

\paragraph{}
Thus, the memory performance of the applications will be evaluated with the total amount of RAM used by the application, computed from the average of a 100 metric collection. For the Progressive Web Application, this measurement includes all processes involved when running the PWA: the Browser, the Renderer and the GPU processes. The application with the least memory consumption will be considered the most memory performant.

\subsection{CPU usage}

There are 2 command lines that can be used to measure CPU usage of mobile applications on Android: \textit{adb shell top} and \textit{adb shell dumpsys cpuinfo}. \textit{Top} is similar to the Linux command of the same name, but with limited options. On Android, it is not possible to change the display from percentage of total CPU available to percentage of CPU core. This makes it less accurate than \textit{dumpsys cpuinfo} which displays the percentage of CPU core. Thus, \textit{dumpsys cpuinfo} will be used to measure CPU usage of mobile applications.

\paragraph{}
The same command line can be used for Progressive Web Apps by adding the CPU usage of the 3 main processes (Browser, Renderer, GPU).
However, some CPU usage measured by dumpsys might not come from the application in itself but from some other tasks done by the browser, or other web applications running on the background.
Thus, other ways to obtain the CPU usage of the application were looked into.
The CPU graph from Chrome Devtools performance panel and the cpuTime computed for each frame were considered inadequate as they compute the self-time of the recorded functions, and not their CPU usage. The CPU sampling events saved during a recording promised more accurate measures \cite{cpu_sampling}. This method for measuring CPU was evaluated against Android's method with a single-core emulated device and a Progressive Web App able to trigger different CPU workloads. \newline

\paragraph{}

```
The CPU performance of the applications will be evaluated with the CPU usage of the
applications while they run, computed from the average of a 100 metric collection.
Because of the results presented in \autoref{annex:cpu_tools}, Android's method
will be used to extract CPU usage which will include all processes involved in the
application (the application package for the mobile applications, the Browser,
Renderer and GPU processes for the PWA). The experiments will be done in airplane
mode to minimize the overhead that the browser might cause.

\paragraph{}
Having a high CPU usage for an application means a higher battery consumption and a
lesser user experience if other applications also require a lot of CPU. The
application with the smallest CPU usage will be considered the most CPU performant.

\subsection{Benchmark}
\label{method:benchmark}

A total of 3 benchmark applications were developed for this project: a Native
Android app, an Interpreted app using React Native and a PWA using ReactJS. The
frameworks React Native and ReactJS were chosen for the Interpreted app and PWA
because of their popularity and their similarity. Their common library 'react' and
common architecture logic reduce the performance difference that can be introduced
by the use of different frameworks.

\paragraph{}
Because of the empirical model of a frame rendering workflow identified and
presented in \autoref{ch:model}, all the applications contain the same features: a
Clicking screen where the user can change the content by clicking on the screen, a
Scrolling screen where the user can scroll the page to see all the content available
and a Home screen where the user can navigate to the Clicking or the Scrolling
screen. The content can be changed from text to picture and vice-versa. The
screenshots of the applications are available in figures \ref{fig:native_screens},
\ref{fig:hybrid_screens} and \ref{fig:pwa_screens}.

\begin{figure}
\centering
\subcaptionbox{Home}{\includegraphics[width=2.1cm]
{kththesis/screenshots/native_home.png}}
\hfill
\subcaptionbox{Clicking - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/native_clicking_text.png}}
\hfill
\subcaptionbox{Clicking - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/native_clicking_picture.png}}
\hfill
\subcaptionbox{Scrolling - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/native_scrolling_text.png}}
\hfill
\subcaptionbox{Scrolling - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/native_scrolling_picture.png}}
\caption{Native application}
\label{fig:native_screens}
\end{figure}

\begin{figure}
\centering
\subcaptionbox{Home}{\includegraphics[width=2.1cm]
{kththesis/screenshots/hybrid_home.png}}
\hfill
\subcaptionbox{Clicking - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/hybrid_clicking_text.png}}
\hfill
\subcaptionbox{Clicking - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/hybrid_clicking_picture.png}}
\hfill
```

```
\subcaptionbox{Scrolling - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/hybrid_scrolling_text.png}}
\hfill
\subcaptionbox{Scrolling - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/hybrid_scrolling_picture.png}}
\caption{Interpreted application}
\label{fig:hybrid_screens}
\end{figure}

\begin{figure}
\centering
\subcaptionbox{Home}{\includegraphics[width=2.1cm]
{kththesis/screenshots/pwa_home.png}}
\hfill
\subcaptionbox{Clicking - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/pwa_clicking_text.png}}
\hfill
\subcaptionbox{Clicking - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/pwa_clicking_picture.png}}
\hfill
\subcaptionbox{Scrolling - text}{\includegraphics[width=2.1cm]
{kththesis/screenshots/pwa_scrolling_text.png}}
\hfill
\subcaptionbox{Scrolling - picture}{\includegraphics[width=2.1cm]
{kththesis/screenshots/pwa_scrolling_picture.png}}
\caption{Progressive Web App}
\label{fig:pwa_screens}
\end{figure}

\paragraph{}
The experiments will test 4 different scenarios: changing a text, changing a
picture, scrolling a text and scrolling pictures. The use of emulators were
considered for the experiments, and their performance was tested against physical
devices. The results of these tests are available on \autoref{annex:emulators}. Due
to the inaccuracy of the emulators, the experiments will only take place on real
devices: a Samsung S6 (Android version 7.0), a Samsung S5 (Android version 5.0) and
a Huawei P9-Lite (Android version 7.0).
\paragraph{}
Because of the tools limitation, clicking will be recorded for 25s and scrolling
for 3s before extracting a memory snapshot and the CPU usage of the recording time.
All the metrics will be measured at the same time for the native and interpreted
applications. For the Progressive Web Apps, the frames and the resources used will
be measured in separate experiments in order to avoid the overhead that the trace
recording might cause.

\paragraph{}
In this chapter, we explored and identified relevant metrics to evaluate the
smoothness, the memory performance and the CPU usage of Mobile and Progressive Web
Apps, and presented the methods used to collect them. For the smoothness performance
of Progressive Web App, that meant building an empirical model of Chrome frame
rendering workflow as well as a tool to collect the frame rendering duration. The
model of a frame rendering workflow, and the smoothness metric will be presented in
details in the next chapter.


\chapter{Chrome's rendering model}
\label{ch:model}
To compare the smoothness performance of mobile and progressive web applications, a
good understanding of their frame rendering workflow is necessary. As no known
document provides this understanding for Chrome, a model of Chrome's frame rendering
workflow was built empirically. This section presents the resulting model and the
frame rendering duration deducted from it.

\subsection{Model of Chrome's frame rendering workflow}
```

```
699 \label{results:chrome_model}
700
701 Following the methodology proposed in \autoref{method:smoothness}, we formulate and
    describe the following model as the one used by Chrome to render new frames in
    Android devices.\newline
702
703 \begin{figure}[h]
704 \centering
705 \includegraphics[width=13cm]{kththesis/Figures/surface_diagram.png}
706 \caption{Surface Diagram}
707 \label{fig:surface_diagram}
708 \end{figure}
709
710 \paragraph{}
711 A frame in Chrome is an aggregation of several surfaces computed by different
    threads (\autoref{fig:surface_diagram}). The App surface is computed by the
    Compositor thread and represents everything displayed by the application. The
    Browser surface computed by CrBrowserMain (referred to as Browser thread)
    represents everything displayed by the browser itself, for example the scrolling
    bar, the refreshing animation or the address bar in regular web applications. There
    can be more than two surfaces aggregated in a frame, for example if there is a video
    inside the web application or embedded ads.
712
713 \begin{figure}[h]
714 \centering
715 \includegraphics[width=\linewidth]{kththesis/Figures/Surface_aggregation.png}
716 \caption{Surface aggregation}
717 \label{fig:surface_aggregation}
718 \end{figure}
719
720 \paragraph{}
721 The VizCompositor thread manages all those surfaces (see
    \autoref{fig:surface_aggregation}). When a new frame is needed in reaction to user
    input or because of animations, it asks the Browser and/or the Compositor thread for
    a new surface and waits for them. Once it received all the new surfaces, it
    aggregates them into a single frame that it sends to the GPU thread. The latter is
    in charge of pushing the graphic buffer of the frame to SurfaceFlinger.
722
723 \paragraph{}
724
725 \begin{figure}
726 \centering
727 \includegraphics[width=\linewidth]{kththesis/Figures/App_surface.png}
728 \caption{Model of the App surface workflow}
729 \label{fig:app_surface}
730 \end{figure}
731
732 The App surface can be computed in two different ways: a fast path, and a complete
    path. When there is only little changes to be made between two consecutive frames,
    the Compositor re-uses the baseline of the previous frame and only changes it
    slightly. This baseline is what we call a Main frame.
733 \autoref{fig:app_surface} represents the main events involved in the computation
    of a new App surface. To summarize, a frame timeline with a new App surface is as
    follows:
734 \begin{enumerate}[ref={Step}\xspace\arabic*]
735 \item \label{timeline:step1} The VizCompositor thread asks for a new frame to the
    Compositor thread.
736 \item \label{timeline:step2} The Compositor thread agrees to compute a new frame and
    schedules it.
737 \item \label{timeline:step3}The Compositor might also ask a new Main frame to the
    Renderer thread.
738 \item \label{timeline:step4}If so, it compiles it. Regarding the pipeline stages
    presented in the background, it computes the Styling, Layout, Compositing, Pre-
```

```
Painting and Painting. When the Renderer thread is finished compiling the main
frame, it commits it to the Compositor.
\item \label{timeline:step5}When the Compositor receives a new Main frame, it
computes Tiling and Raster if necessary with the help of other threads. The Main
frame can then be activated: it replaces the old main frame in the memory of the
Compositor thread and can be used as a baseline for future frames.
\item \label{timeline:step6} Once the deadline scheduled in \ref{timeline:step2} is
up, the frame is drawn. This step is the last stage of pipeline presented in the
background: Drawing. The Compositor generates a CompositorFrame (or surface) and
sends it to the VizCompositor.
\item \label{timeline:step7}The VizCompositor waits for all necessary surfaces,
aggregates them into a single frame and sends it to the GPU thread.
\item \label{timeline:step8}Finally, the GPU thread receives the frame and pushes
the graphic buffer to SurfaceFlinger
\end{enumerate}

The fast path can be used when only small changes occur between two consecutive
frames and no stages present in \ref{timeline:step4} are necessary. It is the case
for example when scrolling or for some CSS animations. The fast path can also be
used when \ref{timeline:step4} takes too long to compute. The Compositor renders a
new frame with fewer changes than planned, and the Main Frame being computed will be
rendered with the next frame.
\paragraph{}
A frame timeline with a new Browser surface is similar to a new App surface. The
main difference is that a Browser surface always follows a complete track and it is
computed entirely on the Browser thread.

\subsection{Synchronizing frames on Android and Chrome}
\label{soundness:metric}

A frame timeline is very different on Chrome and Android.
On one hand, Android's pipeline is synchronous and always follows the same path. The
OS sends VSync signals depending on the device refresh rate, asking everything on
the foreground to render a frame. In response, the UI thread calls
Choreographer\#doFrame which handles callbacks for input and animation before doing
some sizing and layout. The Choreographer then hands what it computed to the
RenderThread which issues the draw commands and swap the buffers for
SurfaceFlinger.

\paragraph{}
On the other hand, Chrome's pipeline is flexible and depends on a lot of parameters.
The VizCompositor also calls the Choreographer regularly after VSYNC signals but
stop the process at animation callbacks. On Chrome's pipeline, the VizCompositor is
also the one to start the frame. One animation callback of the Choreographer might
trigger the start of the frame on VizCompositorThread, but no documentation is
available to confirm this hypothesis and to give the time span between VSYNC signal
and the start of the frame in Chrome. \newline
Thus, it is meaningless to compare Chrome frames with Android frames from the VSYNC
signal.

\paragraph{}
Another method is to compare the time it takes to swap buffers from the moment it
decides to compute a new frame. That means from the start of the Choreographer for
Android and the 'IssueBeginFrame' event for Chrome according to the empirical
model. However, 'IssueBeginFrame' which is considered the start of the frame on the
empirical model might be triggered by other events that were not detected. Moreover,
while Android starts computing the frame immediately, Chrome only schedules the
computation of the frame. Thus, it can take into account events that happened after
scheduling the frame and before computing it.
Therefore, this measure would be too inaccurate on Chrome. \newline


The last possible comparison is the amount of time Chrome and Android spent on
computing the frame before swapping buffers. For Android, it starts with
'Traversal', and can be accessed with the \textit{PerformTraversalStart} timestamp.
```

```
For Chrome, it depends on the type of frame pushed to SurfaceFlinger. We formulate
the following 4 main issues:
765 \begin{enumerate}
766 \item \textbf{When do Chrome starts computing a Main Frame?} \newline
767 As the first stages of Chrome's pipeline happens in \ref{timeline:step6}, when the
Renderer thread comes into play, the beginning of this step will be considered the
start of the computation of a Main Frame.
768 \item \textbf{When do Chrome starts computing a Basic Frame that re-uses a Main
Frame?} \newline
769 The only stage of Chrome's pipeline computed in a Basic Frame is Drawing, which
begins at \ref{timeline:step6}. Thus, it will be considered the start of the
computation of a Basic Frame.
770 %\todo{add cross reference. Done in this case, use the same if you need to add more
from here on}
771 \item \textbf{Do frames which only changes the Browser surface count as frames to
compare to mobile applications on Android?} \newline
772 The Browser surface represents on PWA what is usually also handled by the
application on Android (scrollbar, refresh animation). Thus, those frames will also
be compared to Android frames.
773 \item \textbf{When both the App surface and the Browser surface changes, what to
consider the start of the frame?} \newline
774 Since they are pushed to SurfaceFlinger as a single frame, the start of the frame
will be the start of the Browser or the App surface depending on which happens the
earliest.
775 \end{enumerate}
776
777 \paragraph{}
778 Therefore, in the context of comparing the smoothness of Mobile applications and
PWA, the frame rendering duration is defined in this work as the time Android and
Chrome start actively computing the frame until the graphic buffers are swapped.
779
780 %\todo[inline, color=cyan!20]{Added conclusion to chapter}
781 This chapter deeply described the model of Chrome frame rendering workflow and the
smoothness metric used to compare the smoothness of Mobile applications and
Progressive Web Apps. Besides, we discussed the process to infer the correct events
used to identify the frame rendering timestamps. The next chapter will present the
results of the experiments and answer the research questions.
782
783 \chapter{Results}
784 \label{ch:results}
785
786 This study aims at comparing the smoothness, the memory performance and the CPU
usage of Mobile and Progressive Web Apps. To this end, 3 applications (Native,
Interpreted and PWA) were developed and their performance monitored in 4 different
scenarios: changing a text, scrolling a text, changing pictures, scrolling pictures.
This chapter will describe the results of those experiments, computed from the
average a 100 measurements.
787
788 \label{results:performance}
789
790 \section{Smoothness performance}
791
792 The frame rendering duration presented in \autoref{tab:smoothness} is the average
amount of time the applications compute the frames and send them to
\hyperref[def:surfaceflinger]{SurfaceFlinger}. The smaller this time is, the
faster the application can render frames and the more it can perform other actions
between frames (handle user input, fetch data, optimize computations).
793
794 \begin{table}[h]
795 \centering
796 \resizebox{\textwidth}{!}{
797 \begin{tabular}{|c|c|c|c|c|}
798 \hline
799 & Changing a text & Scrolling a text & Changing a picture & Scrolling pictures \\
800 \hline
```

```
\textbf{Native} & \textbf{13,79} & \textbf{8,54} & \textbf{12,26} & \textbf{7,10} \\
Samsung S6 & 16,50 & 17,32 & 9,66 & 13,36 \\
Samsung S5 & 14,38 & 3,85 & 17,04 & 3,38 \\
Huawei P9-Lite & 10,47 & 4,46 & 10,07 & 4,57 \\
\hline
\textbf{Interpreted} & \textbf{10,13} & \textbf{10,13} & \textbf{7,38} &
\textbf{9,89} \\
Samsung S6 & 13,46 & 18,40 & 9,48 & 17,44 \\
Samsung S5 & 10,43 & 5,19 & 7,83 & 4,62 \\
Huawei P9-Lite & 6,50 & 6,81 & 4,83 & 7,63 \\

\hline
\textbf{PWA} & \textbf{24,03} & \textbf{22,92} & \textbf{15,53} & \textbf{22,64} \\
Samsung S6 & 27,62 & 24,71 & 17,86 & 24,31 \\
Samsung S5 & 21,99 & 21,96 & 15,67 & 21,31 \\
Huawei P9-Lite & 22,47 & 22,09 & 13,07 & 22,31 \\
\hline
\end{tabular}
}
\caption{Average frame rendering duration (ms)}
\label{tab:smoothness}
\end{table}

\paragraph{}
Each column describes the results for one scenario : changing a text, scrolling a
text, changing a picture and scrolling pictures. Each row describes the average
result of each type of application (Native, Interpreted and PWA) and details the
results for each physical device tested. The numbers in bold represent the average
frame rendering duration of the whole cell, that is computed from all devices of the
same application.

\paragraph{}
Firstly, we notice a difference of performance between devices even for the same
scenario and application. This difference is significant especially for the Native
application and the Interpreted application when scrolling a text or pictures
(Columns 2 and 4). The Native application computes a new frame in 17,32 ms on
Samsung S6 and less than 5 ms on Samsung S5 and Huawei when scrolling a text (Column
2), and in 13,36 ms on Samsung S6 and less than 5 ms when scrolling a picture
(Column 4). We observe a similar gap with the Interpreted application, as scrolling
a text (Column 2) results in 18,40 ms frame rendering time on Samsung S6 and
respectively 5,19 ms and 6,81 ms on Samsung S5 and Huawei, and scrolling a picture
(Column 4) results in 17,44 ms of frame rendering time on Samsung S6 and
respectively 4,62 ms and 7,63 ms on Samsung S5 and Huawei. \newline
The difference is less important with the Progressive Web App as the biggest
variation is 5 ms inside the same scenario.

%\paragraph{}
%Firstly, we can notice a difference of performance of the Native and Hybrid
application between devices. In both Scrolling scenarios (Columns 2 and 4) , the
mobile applications take 3 or 4 times more time to compute a frame on Samsung S6
than on Samsung S5 and Huawei. The difference is less significant on Changing
Scenarios (Columns 1 and 3) though a two-times magnitude difference can still be
observed. This difference is not visible on the Progressive Web Application which
offers a more consistent smoothness.

\paragraph{}
Secondly, it was expected that the scrolling scenarios (Column 2 and 4) would result
in smaller frames rendering duration in Progressive Web Apps as the 'fast path' is
taken more often. This is true when displaying text (Columns 1 and 2), though the
difference is small (1,11 ms difference between changing and scrolling), but not at
all when displaying pictures since scrolling pictures (Column 4) results in 22,64 ms
of average frame rendering duration compared to 15,53 ms when changing a picture
(Column 3).

```

```
%\paragraph{}
%Lastly, the average frame duration of the Progressive Web App is well above the Native and Hybrid applications in every scenarios except when changing pictures (Column 3) on Samsung S5 as the Native app takes 17,04 ms and the PWA 15,67ms. The difference can be significant and attain 10 ms and more in several cases.
\paragraph{}
Lastly, we observe that the average frame duration of the Progressive Web App is well above the Native and Interpreted applications in every scenario. The difference varies from 10 to 15 ms when changing a text, scrolling a text or scrolling a picture (Columns 1, 2 and 4), and goes down to 3 ms when changing pictures (Column 3).

\paragraph{}
In conclusion, the PWA took to compute more time to render new frames than the Mobile applications, and often by 2 or 3 times the amount of time it took for the Native and Interpreted App. Thus, we can conclude that Progressive Web Application are not as smooth as Mobile Applications.

\section{Memory performance}

The memory performance is an important performance parameter in mobile applications as this memory is limited. It is evaluated using the amount of RAM used by the applications just after the running a scenario. We can make several observations from the results presented in \autoref{tab:memory}.

\begin{table}[h]
\centering

\resizebox{\textwidth}{!}{
\begin{tabular}{|c|c|c|c|c|}
\hline
& Changing a text & Scrolling a text & Changing a picture & Scrolling pictures \\
\hline
\textbf{Native} & \textbf{55,4} & \textbf{60,5} & \textbf{66,1} & \textbf{102,3} \\
Samsung S6 & 68 & 77 & 87 & 106 \\
Samsung S5 & 47 & 49 & 56 & 104 \\
Huawei P9-Lite & 50 & 55 & 56 & 97 \\
\hline
\textbf{Interpreted} & \textbf{202,5} & \textbf{374,6} & \textbf{293,8} & \textbf{491,5} \\
Samsung S6 & 196 & 377 & 300 & 578 \\
Samsung S5 & 169 & 341 & 265 & 283 \\
Huawei P9-Lite & 243 & 406 & 317 & 614 \\
\hline
\textbf{PWA} & \textbf{227,0} & \textbf{190,3} & \textbf{286,1} & \textbf{193,2} \\
Samsung S6 & 257 & 179 & 349 & 182 \\
Samsung S5 & 148 & 243 & 193 & 243 \\
Huawei P9-Lite & 276 & 149 & 316 & 154 \\
\hline
\end{tabular}
}
\caption{Average RAM consumption (MB)}
\label{tab:memory}
\end{table}

\paragraph{}
Each column describes the average RAM consumption for one scenario : changing a text, scrolling a text, changing a picture and scrolling pictures. Each row describes the average result of each type of application (Native, Interpreted and PWA) and details the results for each physical device tested. The numbers in bold represent the average RAM consumption of the whole cell, that is computed from all devices of the same application.
```

```
\paragraph{}
The first observation is the wide gap between the memory used by the Native Application, and the Interpreted and Progressive Web App. The Native application consumes at most 106 MB across all devices and scenarios, while the PWA and Interpreted app consumes at least 148 MB and on average 190 MB to 491 MB of RAM.

\paragraph{}
As expected, displaying text (Column 1 and 2) consumes more memory than displaying picture (Column 3 and 4) for every scenario and device. The applications consume around 10.7 MB (Native), 91.3 MB (Interpreted) and 59.9 MB (PWA) more memory when changing pictures than when changing texts. Scrolling requires 41.8 MB (Native), 116.9 MB (Interpreted) and 2.9 MB more memory when the content are pictures rather than text.

\paragraph{}
Scrolling content consumes more memory than changing it both for the Interpreted and Native application. We observe a difference of 5.1 MB (Native) and 172.1 MB (Interpreted) when the content is text, and 36.2 MB (Native) and 197.7 MB (Interpreted) when the content is picture. However, it is the other way around for the Progressive Web App as scrolling text requires 36.7 MB less RAM than changing text, and scrolling pictures requires 92.9 MB less RAM than changing pictures.

\paragraph{}
Lastly, we notice that the PWA consumes less memory than the Interpreted application, except when changing text. However, the difference when changing text is quite small, 24.5 MB compared to other scenarios, 184.3 MB (scrolling text), 7.7 MB (changing pictures) and 298.3 MB (scrolling pictures).

\paragraph{}
In conclusion, the Progressive Web App consumes more RAM than the Native application, but less than the Interpreted application. Thus, we can conclude that the memory performance Progressive Web App, though still far from Native applications, do compare to Interpreted applications and hence mobile applications.
%\paragraph{}
%The second observation is the memory used in each scenarios. As expected, displaying pictures (Column 3 and 4) requires more memory than text (Column 1 and 2), either when changing content or scrolling. Scrolling consumes more memory than changing content on both the Native and the Interpreted applications, and on the Progressive Web App on Samsung S5. However, it is the other way around for the PWA on Samsung S6 and Huawei P9-Lite: changing the content requires more memory than scrolling.

%\paragraph{}
%The last observation is the comparison between the Interpreted and Progressive Web Application. The difference between them can be significant in the scrolling scenarios with the Interpreted consuming twice as much memory as the PWA, sometimes more. It is marginal in the changing content scenarios with the PWA consuming slightly more than the Interpreted application.

%\paragraph{}
%The Progressive Web App consumed more memory than the Native application in every scenario, with the difference ranging from 58\% to 464\%. However, the Interpreted application consumed similar amount of memory than the PWA when changing content, and a lot more than the PWA when scrolling. Hence, we can conclude that though the memory performance of Progressive Web Apps do not compare to Native applications, it is similar or even better than Interpreted applications depending on the interaction.


\section{CPU Usage}

The CPU on mobile phones is a limited resource. The more an application consumes, the less work can be done on the background and the more energy it consumes. We can make several observations from the results presented in \autoref{tab:cpu}

\begin{table}[h]
```

```
\centering

\resizebox{\textwidth}{!}{
\begin{tabular}{|c|c|c|c|c|}
\hline
& Changing a text & Scrolling a text & Changing a picture & Scrolling pictures \\
\hline
\textbf{Native} & & & & \\
Samsung S6 & 7,97 & 40,13 & 9,09 & 34 \\
Samsung S5 & 3,63 & 28,57 & 8,41 & 37,21 \\
Huawei P9-Lite & 4,03 & 20,18 & 8,99 & 51,13 \\
\hline
\textbf{Interpreted} & & & & \\
Samsung S6 & 59,59 & 90,03 & 57,82 & 106,68 \\
Samsung S5 & 44,89 & 59,42 & 44,36 & 60,35 \\
Huawei P9-Lite & 31,44 & 73,55 & 32,09 & 116,37 \\
\hline
\textbf{PWA} & & & & \\
Samsung S6 & 28,28 & 105,63 & 25,66 & 102,65 \\
Samsung S5 & 15,09 & 66,86 & 11,56 & 66,35 \\
Huawei P9-Lite & 22,35 & 105,33 & 19,36 & 119,17 \\
\hline
\end{tabular}
}
\caption{Average CPU usage (\% CPU core)}
\label{tab:cpu}
\end{table}

\paragraph{}
Each column describes the average CPU consumption for one scenario : changing a
text, scrolling a text, changing a picture and scrolling pictures. Each row
describes the average result of each type of application (Native, Interpreted and
PWA) and details the results for each physical device tested. As \% CPU core is a
unit unique to each device, the average \%CPU core of all devices was not computed
this time.

\paragraph{}
Again, the first observation is the difference of performance between the Native
application and the Interpreted app and PWA. On one hand, the Native application
consumes less than 10\% CPU when changing the content (maximum of 7.97\% when
changing text and 9.09\% when changing pictures), and no more than 50\% CPU when
scrolling (maximum of 40.13\% when scrolling text and 51.13\% when scrolling
pictures). On the other hand, the Interpreted application consumes at least 31.44\%
CPU when changing content and 59.42\% when scrolling. The Progressive Web App
consumes between 11.56\% and 28.28\% CPU when changing content, and between 66.35\%
and 119.17\% when scrolling.

\paragraph{}
As expected, all applications consumes more CPU when scrolling (Column 2 and 4) than
when changing content (Column 1 and 3). We notice a difference of at least 14.53\%
CPU core between scrolling and changing text on the same application and device, and
at least 15.99\% CPU core between scrolling and changing pictures.

\paragraph{}
Surprisingly, displaying pictures (Column 3 and 4) does not always consume more CPU
than displaying text (Column 1 and 2). The Progressive Web App consumes between
2.62\% and 3.46\% more CPU core when changing pictures (Column 3) than when changing
text (Column 1). We also observe this phenomenon with the Interpreted app on Samsung
S6 (1.77\% more CPU core when changing text than picture) and Samsung S5 (0.53\%
more CPU) and with the Native application on Samsung S6 where scrolling text
consumes 6.13\% more CPU than scrolling pictures.
```

```
%However, displaying pictures does not always consume more CPU than displaying text. We can see pictures consuming less than text on the Interpreted application when changing content, though the difference is small, and especially on the Progressive Web App.

\paragraph{}
The last observation concerns the comparison between the Interpreted and Progressive Web Application. One one hand, the PWA consumes noticeably less CPU than the Interpreted app when changing content. The difference between the Interpreted app and the PWA ranges from 9.09\% to 31.31\%CPU consumption on the same device when changing text, and from 12.77\% to 32.16\% CPU consumption when changing pictures. On the other hand, the PWA usually consumes more CPU than the Interpreted app when scrolling. Scrolling text requires between 7.44\% and 31.78\% more CPU with the PWA than with the Interpreted app. The difference when scrolling pictures is less important, as the PWA consumes 2.80\% more CPU on the Huawei and 6.08\% more CPU on the Samsung S5 than the Interpreted app, but 4.03\% less CPU on the Samsung S6.

%\paragraph{}
%The last observation concerns the comparison between the Interpreted and Progressive Web Application. One one hand, the PWA consumes noticeably less CPU than the Interpreted app when changing content, often half as much. On the other hand, it consumes slightly more when scrolling content though the difference is usually than 10\%.

\paragraph{}
The conclusion to the CPU usage of Mobile and Progressive Web App is similar to the memory consumption. In every scenario, the Progressive Web App consumed more CPU than the native application though the difference is not as big as the memory consumption. However, it also consumed similar amount of CPU than the Interpreted application when scrolling, and less when changing content. Thus, Progressive Web Apps can be considered better than Interpreted applications in terms of CPU usage, but worse than Native applications.

\paragraph{}
This chapter presented the results of the experiments conducted to compare the smoothness, memory performance and CPU usage of Progressive Web Apps and Mobile applications. The next chapter will discuss those last results and the limitations of the conclusions.
%To summarize the results presented in this chapter, it was found that the CPU usage measured by Chrome tools is not as accurate as Android tools and that Android emulators are not precise enough to use them in CPU experiments. A smoothness metric able to compare Mobile applications and Progressive Web app was defined and used to compare a Interpreted, a Native and a Progressive Web Application. The next chapter will discuss those results.


\chapter{Discussion}
\label{ch:discussion}

In the last chapter, we presented several results about the CPU sampling method of Chrome DevTools, the model of Chrome frame rendering workflow built empirically, the smoothness metric defined according to the model and the evaluation of the smoothness, the memory performance and the CPU usage and Mobile and Progressive Web Apps. This chapter will discuss those results and present their limitations.
%The last chapter presented the results of several experiments regarding measurements of CPU for PWA and smoothness performance. This chapter will discuss those results and their limitations.


\section{Soundness of the model}
The smoothness metric used to compare Mobile and Progressive Web Applications was defined using an empirical model of Chrome's rendering pipeline. Though this model is not usable for every web applications, for example with videos or third-party advertisements, it is consistent with the events observed in simple web applications and allowed a comparison of the frames computed by Chrome and by Android framework. Since the start of a frame is not triggered by the same events on Chrome and
```

Android, the frame duration is defined as the time it takes to compute a frame and push it to SurfaceFlinger. However, this metric was identified only for applications running with Chrome browser on Android. Other smoothness metrics might be used with other browsers and other OS.

%\section{CPU usage of Progressive Web App}
%We identified 2 different methods to evaluate the CPU usage of Progressive Web App running in Chrome: the CPU sampling method of Chrome DevTools and the CPU usage of all processes involved in the Progressive Web App measured by \textit{dumpsys cpuinfo}. We found that the measurements of \textit{dumpsys} were more accurate than Chrome DevTools despite the overhead caused by other tasks not related to the PWA. Though this overhead was minimized by conducting the experiment in airplane mode, with no WiFi and no other web app running in the background, it is still present in the results.

\section{Experiments}

Some small issues were raised during and after the experiments.\newline
Firstly, the scrolling experiments were not as consistent between frameworks and devices as changing content was. The same drag event did not trigger the same scrolling speed, and the length of the scrolling page was different. The tools used to simulate drag events also behaved differently. Thus, the drag events were adapted to the device and framework so that the view scrolled all the way down and a little up during the experiments. \newline
Lastly, the clocks of different threads in Chrome were sometimes found to be slightly out of sync. Thus, some events were detected before their triggering events on another thread. However, those incidents concerned events not counted in the frame duration. They were scarce and the delay was of small magnitude. Thus, it was concluded that they did not impact the results presented here.

\section{Smoothness}

The smoothness performance of native, interpreted and progressive web applications was evaluated in 4 different scenarios as they triggered different stages of Chrome's rendering pipeline: changing a text, scrolling a text, changing pictures and scrolling pictures. In almost every scenario, the smoothness of the Progressive Web App was found to be lagging behind those of native and interpreted applications, often by a 2-times magnitude.\newline
Contrary to what was expected, the frame computation time of Progressive Web Apps when changing a picture was a lot smaller than when scrolling pictures. As this is not observed when the content is text, this might be due to a better usage of the GPU, though this is only a hypothesis. \newline
The Native and Interpreted application have similar results. The Interpreted app is faster at computing frames when changing the content displayed than the Native application, and slower when scrolling the view. As the difference is only of 2 or 3 ms during a scroll, and can attain 10 ms when changing content, the Interpreted application can be considered to be smoother than the Native application.

\section{Memory and CPU}

The resources measured during the experiments were the CPU and Memory usage. The Native application was found to consume a lot less resources than the Interpreted and Progressive Web Applications. This will more likely impact the battery consumption, and corresponds to the result found by Ciman and Gaggi \cite{ciman2017empirical} who compared the energy consumption of Native and Cross-platform applications. They concluded that cross-platform applications consume more energy than native applications, especially when updating the User Interface. However, it does not concur the results found by Tjarco \cite{PWAapplicability} who compared the energy consumption of a Progressive Web App and Native applications and found that PWA consume less energy. Nevertheless, the applications used for his experiments also accessed the network regularly, impacting the results. Moreover, though a high CPU and Memory usage does consume more energy, it is not an accurate indicator of energy consumption.

\section{Limitations}

\paragraph{}
One of the main limitation of the results presented previously lies in the number of devices used for the experiments. As was observed in \autoref{results:performance}, the performance of the applications can differ greatly between devices. More

```
experiments on other devices need to be conducted to validate the conclusion reached
from the current results. \newline
995 A second limitation is the measurements of the resources used the Progressive Web
app. As was mentioned in \autoref{results:cpu}, those measurements are less
accurate than for the mobile applications because the browser may perform other
tasks unrelated to the PWA. Those tasks were minimized during the experiments but
can still impact the results. \newline
996 Another limitation comes from the limited cross-platform and Web framework used. The
results presented here only applies to React Native and React as every framework has
its own strength and weaknesses.
997 The last limitation comes from the OS and browser studied during this project. Only
Android and Chrome browser was studied even though other browsers and OS are used
regularly by numerous people. Those other browsers and OS, especially Safari on iOS
need to be studied before validating the conclusion reached during this work on the
performance of Progressive Web Application compared to mobile applications.
998
999 \paragraph{}
1000 The results achieved during this work were discussed as well as their limitations.
The next chapter will conclude this work.
1001
1002
1003 \chapter{Conclusion}
1004 \label{ch:conclusion}
1005
1006 This chapter concludes this study of Progressive Web Applications. Firstly, we will
summarize the work done during this study. Then, we will suggest possible future
work related to Progressive Web Apps.
1007
1008 \section{Contributions}
1009
1010 This study aimed at answering 3 research questions :
1011 \begin{itemize}
1012 \item \textbf{RQ1:} How does the smoothness of Progressive Web Apps compare to
Mobile applications ?
1013
1014 \item \textbf{RQ2:} How does the memory performance of Progressive Web Apps compare
to Mobile applications?
1015 \item \textbf{RQ3:} How does the CPU usage of Progressive Web Apps compare to Mobile
applications?
1016 \end{itemize}
1017
1018 %The main research question revolves around the smoothness performance of
Progressive Web Applications compared to Mobile Applications on Android. The
smoothness performance refers in this work to how well frames are computed, and the
efficient use of resources to do so.
1019
1020 \paragraph{}
1021 As the smoothness of an application is defined as how fast it can render frames, we
needed tools able to precisely give the start of end of a frame on Chrome and
Android. However, at the time of writing no such tool existed for Chrome. Hence, we
build a model of Chrome frame rendering workflow and developed a tool,
\textit{FrameTracker}, able to track the frames from the beginning until the end.
With this, we were able to define a smoothness metric able to compare the frames of
Android and Chrome.
1022
1023 \paragraph{}
1024 As the use of \textit{dumpsys cpuinfo} for Progressive Web Apps could include some
overhead, we evaluated another method able to measure the CPU usage of PWA: the CPU
sampling method of Chrome DevTools. However, it was found that it was less accurate
than \textit{dumpsys cpuinfo} despite the overhead.
1025
1026 \paragraph{}
1027 Finally, we developed 3 applications: a Native, a Interpreted and a Progressive Web
App and evaluated their smoothness, memory performance and CPU usage during 4
different scenarios: changing and scrolling text and pictures. \newline
```

All experiments were automated in order to remove the human interaction variable from the results and make them reproducible. This was done using \textit{monkeyrunner} for experiments with \textit{dumpsys} and Chrome Devtools Protocol experimental methods \textit{Input.synthesizeTapGesture} and \textit{Input.synthesizeScrollGesture} for experiments with trace recording. The experimental \textit{Tracing} domain also allowed the automation of Chrome Trace recordings.

\paragraph{}

The results showed that the Progressive Web App was not as smooth as Mobile Applications on Android. However, it had similar or better CPU usage and memory performance than the Interpreted application depending on the scenario, though it was less performant than the Native application. \newline

Nonetheless, those results do not mean developers should forget about Progressive Web Apps. The impact of the difference of performance on the end-users remains to be studied. Moreover, this study focused on the performance of PWA compared to Mobile Applications. It did not examine the benefits or drawbacks of Progressive Web Apps against regular Web Apps.

\iffalse

Then, we evaluated the smoothness of a Interpreted, a Native and a Progressive Web Application and found that the Progressive Web App was not as performant as the Mobile applications.

\paragraph{}

First, a model of Chrome's rendering pipeline was built empirically. This model allowed us to define a smoothness metric able to compare the frame duration of Mobile Applications on Android and Progressive Web Apps on Chrome. Since, the event triggering a new frame on Chrome is difficult to identify, and new events can impact the frame until late on Chrome's rendering pipeline, this metric was defined as the computation time of the frame before it is handled by the display manager of Android.

\paragraph{}

This metric allowed us to compare the frame computation time of Mobile Applications and Progressive Web Apps with different user interactions and different type of content displayed. In all scenarios, the Progressive Web Application was a lot slower to compute a new frame than the Native and Interpreted Applications. Thus, we can conclude that Progressive Web Apps are not as smooth as Mobile Applications on Android.

\paragraph{}

The resources used, namely the CPU and the Memory were also measured to compare the efficiency of the applications when computing new frames. In all scenarios, the Native application was more efficient than the Interpreted and Progressive Web Application, both in terms of CPU and Memory. However, the efficiency of the Interpreted and the Progressive Web Application was similar. From this, we can infer that progressive Web Applications does not consume more resources than Interpreted applications, but Native Android Applications remain more efficient.

\paragraph{}

To summarize, we found that Progressive Web Applications are not as smooth as Mobile Applications on Android. Their resource consumption, though similar to Interpreted applications, is higher than Native Applications. With this, we can conclude the smoothness performance of Progressive Web Apps is not as smooth as Native and Interpreted Applications on Android.

\fi

\section{Future work}

Several questions remains on the subject of Progressive Web Apps. One of them is the comparison of Progressive Web Application to Mobile applications on iOS. As the support of PWA on iOS is lagging behind Android, this comparison was often overlooked. \newline

Though this work concluded on the poor smoothness of PWA compared to Mobile applications, the impact on the user experience remains to be studied in depth.

The work conducted here on smoothness performance can also be extended to other browsers, other platforms and other cross-platform technologies. This work can also be extended to include the responsiveness of PWA and Mobile Applications, as it is an important aspect of the User Interface Performance. \newline

```
1051 Other aspects of Progressive Web Apps can also be compared to Mobile applications,
     such as the development and maintenance efforts of the developers, and the
     engagement of the end users. \newline
1052 The security aspect is also important to consider. Though Jiyeon et al.
     \cite{Pride_Prejudice} already opened the question, some questions remains, for
     instance the additional security flaws that threatens the integrity of Progressive
     Web Apps compared to regular web applications, and the possibility for end-users to
     install a malicious Progressive Web App as no organization inspect them.
1053 Finally, as Progressive Web Applications can also be installed on desktop, they can
     also be compared to desktop applications. All the research conducted and suggested
     until now to compare them to Mobile Applications can be transferred to desktop
     applications. Though computers often have much almost unlimited resources than
     compared to smartphones, the resource's consumption still need to be studied as
     laptop do not always have an unlimited power source, and some have really limited
     CPU and memory.
1054
1055
1056 \listoftodos{Notes}
1057
1058 \listoffigures
1059 \printbibliography[heading=bibintoc]
1060
1061 \appendix
1062 \chapter{Model of Chrome Rendering pipeline}
1063 \label{annex:chrome_model}
1064
1065 \begin{center}
1066 \includegraphics[angle=90, height=17cm]{kththesis/Figures/Chrome_Graphics_v2.png}
1067 \end{center}
1068
1069 \chapter{Collecting CPU usage of Progressive Web Apps}
1070 \label{annex:cpu_tools}
1071 Since the CPU usage of PWA measured by \textit{dumpsys cpuinfo} may contain
     browser tasks that have nothing to do with the PWA, other measurement methods
     provided by Chrome were considered. One of them is the CPU sampling method executed
     by Chrome DevTools performance panel during a recording.
1072 Different CPU workloads were simulated with a function called at different time
     intervals. The CPU usage for each workload is the average of 100 measurements. The
     results are presented in \autoref{fig:cpu_usage}.
1073
1074 \begin{figure}[h]
1075 \centering
1076 \includegraphics[width=13cm]{kththesis/Figures/cpu_graph.JPG}
1077 \caption[Measurements of CPU Usage of PWA]{Measured CPU usage (\%) over CPU
     workload represented by time laps between function calls}
1078 \label{fig:cpu_usage}
1079 \end{figure}
1080
1081 As was expected, the CPU usage of the Renderer process (where the application lives)
     and the GPU Process decrease with the frequency of the function call. The CPU usage
     of the Browser process, though small, also decreases. This is surprising as the
     function call used to emulate CPU workloads did not involve the Browser in any way.
     This indicates that the Browser process is necessary to run the PWA and not only at
     launch-time, to make network requests or to display a graphic element belonging to
     the Browser.
1082
1083 \paragraph{}
1084 The curve of the CPU usage measured by Chrome's CPU sampling is closest to the curve
     of the CPU usage measured by \textit{dumpsys cpuinfo}. This indicates that it is a
     good estimation of the CPU usage of Chrome. However, it surpasses the measurements
     taken by \textit{dumpsys cpuinfo}. As the latter takes its measurements from the
     system files which count every tick of the CPU, it gives an accurate measurement of
     the CPU usage of Chrome. The Progressive Web App which is run by Chrome cannot
     exceed its CPU usage. Thus, the CPU sampling method of Chrome is less accurate than
```

```
\textit{dumpsys cpuinfo} to measure the CPU usage of a PWA. Later measurements of
CPU will all be given by \textit{dumpsys cpuinfo}.
1085
1086
1087 \chapter{Emulators}
1088 \label{annex:emulators}
1089
1090 Since emulators can provide easy access to a large range of Android devices, their
use was considered for the final experiments. The CPU, of which the usage is a key
metrics used to evaluate the performance of the applications, is a hardware
difficult to emulate \cite{cpu_emulator}. Thus, it is important to test the
emulators against a physical device before using them in CPU-related experiments.
1091
1092 As was presented in the background, only a few Android emulators try to simulate
accurately the hardware capabilities of the emulated device and not improve them.
Only 3 were identified and tested: Android Emulator, Genymotion and Visual Studio
Android Emulator. As was explained previously, the metrics used to compare the
performance of the emulators were the CPU usage using \textit{dumpsys cpuinfo},
the total RAM used using \textit{dumpsys meminfo} and the percentage of janky
frames using \textit{dumpsys gfxinfo}. The measures were taken while the devices
ran the same application that changed a text every few milliseconds. No human
interaction was necessary. The results presented in \autoref{tab:emulators_test}
are the average of 110 measurements.
1093
1094 \begin{table}[!ht]
1095 \resizebox{\textwidth}{!}{
1096 %\centering
1097 \begin{tabular}{|m{2,5cm}|m{2cm}|m{2cm}|m{2cm}||m{2cm}|m{2,5cm}|}
1098 \hline
1099 & \multicolumn{3}{c||}{Samsung S6 (API 7.0)} & \multicolumn{2}{c|}{Samsung S5 (API
5.0)} \\
1100 \hline
1101 Metrics & Physical device & Genymotion & Android Emulator & Physical device & Visual
Studio Emulator \\
1102 \hline
1103 CPU Usage (\%) & 72 & 29 & 18 & 48 & 16 \\
1104 \hline
1105 RAM (KB) & 72 080 & 24 243 & 17 499 & 56 813 & 16 992 \\
1106 \hline
1107 Janky frames (\%) & 13 & 46 & 6 & 3 & 98 \\
1108 \hline
1109 \end{tabular}
1110 }
1111 \caption{Emulators Tests}
1112 \label{tab:emulators_test}
1113 \end{table}
1114
1115 \paragraph{}
1116 The results between the physical devices and the emulators are too different to be
used in future experiments. Thus, only the available Android smartphones will be
used for this project: Samsung S6 (Android version 7.0), Samsung S5 (Android version
5.0) and Huawei P9-Lite (Android version 7.0)
1117
1118
1119 % Tailmatter inserts the back cover page (if enabled)
1120 \tailmatter
1121
1122 \end{document}
```