How Can Web Developers Optimise for Lower-Power Smartphones in the Developing World?

Edward George Prince

6151713

Supervised by Dr Norlaily Yaacob

Computer Science, Department of Computing

29th April 2019

Certificate of Ethical Approval

Applicant:

Edward Prince

Project Title:

How Can Web Developers Optimise for Lower Power Smartphones in the Developing World?

This is to certify that the above named applicant has completed the Coventry University Ethical Approval process and their project has been confirmed and approved as Low Risk

Date of approval:

27 February 2019

Project Reference Number:

P87774

Abstract

This paper looks at methods for optimising modern web applications for users with low-powered smartphones. By analysing a testbed application across multiple JavaScript frameworks and bundlers, it is discovered that performance optimisations can be made without sacraficing user or developer experience. This was found using the Google Development Tools to analyse an app built with React, Node.js, Socket.io and Express.js.

Acknowledgements

A special thanks to Dan Prince for his unwavering encouragement and support in this project.

Contents

1 Introduction

When developing for mobile phones, the devices to which web apps are optimised are the current standard of modern mobile phones. Whilst this proves useful for a target audience of users able to buy this standard of device - it is less so for the many people who are unable to access web apps on new smarthones. Much of the developing world uses mobile phones several generations behind what is in popular usage across developed countries. With Google, Mozilla, Microsoft and Huawei all releasing cheap smartphones for developing countries (Hiscott 2016), it is likely the developing world's mobile device population will follow a trend of being considerably less powerful than the mobile device population of developed countries (Hiscott 2016). With over 98% mobile phone adoption in developing countries (Sharwood, 2017) this offers a large market of potential users for web applications.

With the anticipation of a high speed network and low-power CPU future (more detail in chapter 2), it is important that applications are designed to run on lower-powered CPU's. To test this, this research will include analysis of performance changes when the CPU power is being throttled. Older devices also means a lack of compatibility with the newest browsers, meaning that for global usage, developers must make their applications work on older versions of browsers. To mitigate a loss of quality of the application, the tests should be performed on an application built with modern front-end frameworks, as a realistic benchmark. Then metrics can be gathered by measuring performance of applications even when converted into older versions of JavaScript that are compatible with older browsers - and making sure the application runs on those versions.

These objectives will be met by using the Google Chrome Devtools to monitor the CPU-throttled performance, and different technology implementations that allow comprehensive browser compatibility. There is an expectation that the lighter front-end frameworks perform best.

1.1 Dissertation Structure

The Literature Review explores a selection of the relevant existing literature surrounding and encapsulating the topic, establishing its context. Due to the subject being in its infancy in relation to most areas of Computer Science - there is a distinct lack of relevant literature. Network-based optimisations are a popular article subject - but as it bears little need in developed countries - optimisations for low-powered CPU's is not a thriving research topic.

The Methodology chapter lays the ground work for the experiments to be conducted. It shows the techniques used to gain data, explains how they work, and justifies their selection for this project. The experiments are split up into component sections, each identifying a different area for potential optimisation. Each section has a brief summary, a prediction of the expected results upo

modification and re-profiling, and the actual results, how the compare, and why they might differ to the predictions. The chapter is concluded with a summary of all of the results, and how they relate to optimisations for low-powered smartphones.

The Design and Implementation chapter describes exactly how the experiments were conducted, along with the results. The experiments are split into two case studies, analysing bundler performance, and analysing front-end JavaScript framework performance.

The Discussion chapter begins to weave the results into an argument which responds to the initial research question, along with providing assessment on the quality and validity of this research.

The Project Management chapter provides clarity and reflection on the process of creating this project, and the methodologies employed to ensure completion within the timeframe given.

The Conclusion highlights the extent to which the findings provide a suitable answer to the initial research question. It also highlights further areas for study regarding this research.

1.2 Terminology

Term	Definition
Devtools	A much longer definition than the term itself, even longer now
CDN	Content Delivery Network
ES5	ECMAScript 5
Polyfill	Some
Babel	Compiler

2 Literature Review

2.1 Introduction

As of 2018, there were 3.7 billion mobile devices connecting to the internet (Stevens, 2018) showing that a huge proportion of people use their mobile devices, and there's nothing to suggest that the growth in mobile usage will do anything other than increase.

Given that we know 90% of users stop using an app due to poor performance, and 86% will uninstall the app altogether (AppDynamics and University of London, 2014), there is the possibility for unoptimised web apps to lose huge numbers of potential users due to the app performance not meeting our standard usability criteria.

Nejati and Balasubramanian discuss the difficulty in isolating the effects of the mobile browser in determining the root of poor performance (Nejati, Balasubramanian, 2016). This project will differ from that research by making use of the Google Chrome Dev Tools mobile simulator - taking many of the factors out of consideration, allowing isolation on only changes to the CPU performance. They also identify the issue of mobile browsers being much slower than desktop, as agreed with in other research on mobile optimisations in the cloud era (Wang et al. 2013), a problem given the dramatic increase in mobile devices globally.

The project will also rely on prior research on optimisation of images, (Thiagarajan et al. 2012) - leaving the focus to be on the fundamental pillars of web development - HTML, CSS and JavaScript.

2.2 CPU

Why are Web Browsers Slow on Smartphones? [Criticism checklist] It is suggested that network round-trip time is a major problem. Whilst this may have been true at the time, due to the increase in network capacity, and the trends shown so far, it could be assumed that time-based network problems become increasingly less relevant, with the hardware-based issues becoming more prominent. Particularly in developed countries, modern smartphones with the latest hardware are readily available, which is why this project focuses on mobile phones in usage in the developing world.

Over the last 20 years, Moore's Law has begun to plateau (Simonite, 2016), and hardware capability increases along with it.

Networks on the other hand are continuing to get faster and offer better coverage. Taking Kenya (a developing country) as an example, figure 2 shows that network speeds are increasing at a dramatic rate:

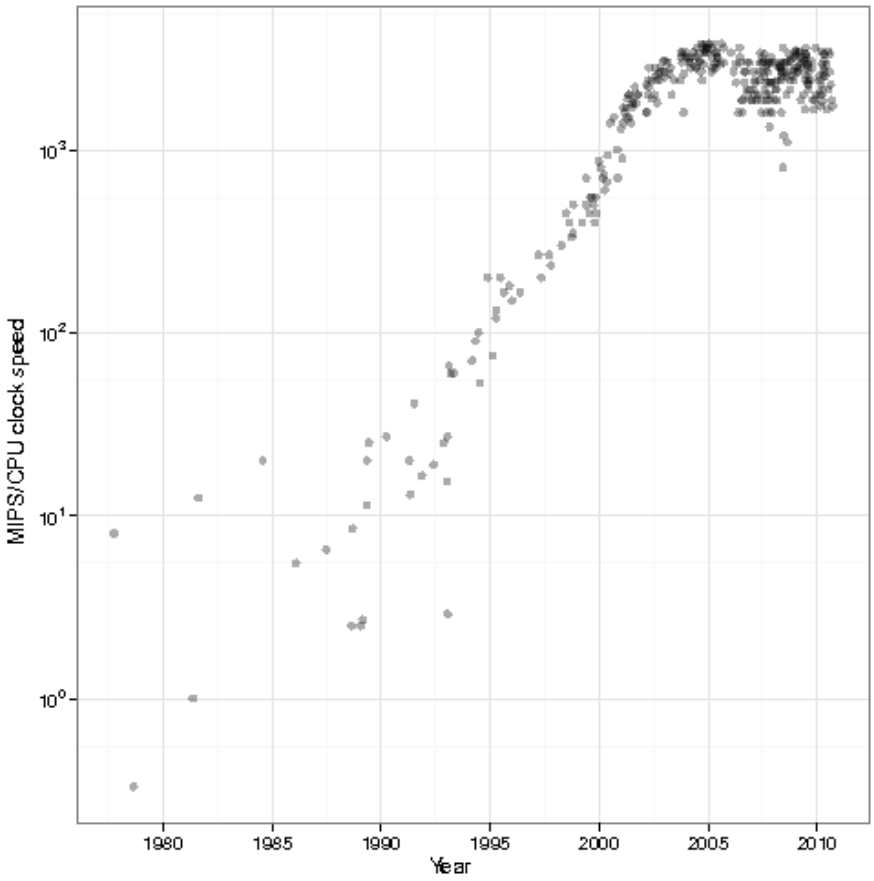


Figure 1: Moore's Law Plateau

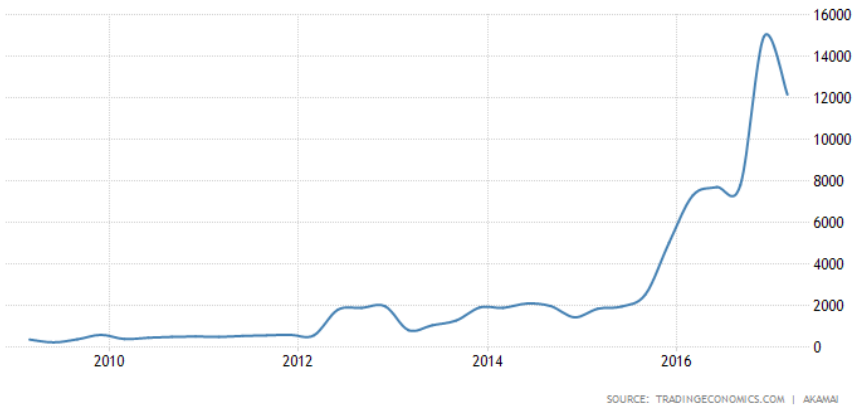


Figure 2: Kenyan Network Speeds

"Of all ITU [International Telecommunication Union] regions, the strongest growth was reported in Africa, where the percentage of people using the Internet increased from just 2.1% in 2005 to 24.4% in 2018" (ITWeb, 2018). Containing a large number of developing countries, by looking at Africa as a whole, and Kenya as a microcosm - we can see there is a likelihood of a fast network future - and with the plateau in Moore's Law, a likelihood of stagnation in CPU performance increases. A fast network future may lead to an emergence in prioritisation of CPU-based optimisation, over decreasing app sizes - which would have an increasingly negligible effect as network speeds improve. There has also been extensive research on optimization over networks already - reducing content, using CDN's, compressing content, utilising browser caching techniques (Iliev, 2014), and front end optimisations for modern devices. This research will avoid liminality by relying on the research already conducted on networks, and instead fill the gap left for optimising web applications to perform highly for low-end smartphones.

An In Depth Study of Browser Performance This article demonstrates research around locating mobile browser bottlenecks. It concludes that computational activities are the main bottlenecks, highlighting the need for research on optimising for lower-powered CPU devices, where the effects of computational bottlenecks will be be exceedingly more relevant.

It also states that this is not the case for desktops, where network activity is the dominant bottleneck - also emphasizing the importance of optimising CPU-intensive applications for mobile devices. Wang et al. suggest that the most costly area for TTI is resource loading - which, as the sources are credible, makes sense - however no speculation is offered about the state of the future of processors and networks, making the research relevant currently, but perhaps not in years to come.

2.3 Mobile Web Browser

Mobile Web Browser Optimisations in the Cloud Era [criticism checklist] There are also many comparisons made between the mobile browser and desktop browser, over the same network speed - finding that the mobile version can take up to nine times as long to load. As they are on the same network - we know that this is therefore down to the hardware and mobile browser - not server-to-client issues. Suggestions have been made that more effort should be put into making mobile browser optimisations - with which future developers could utilise to make web applications more globally usable.

2.4 Development Issues

Analyzing Mobile Browser Energy Consumption [Criticism checklist] Shows that reducing JavaScript and utilising HTML functionality leads to a less power-hungry application. JavaScript being the most power-hungry part of the traditional web application. It also shows the importance of optimising CSS rules, with a reduction to only necessary rules, along with the migration of multiple CSS files into one file for a given page seeing a drop of 5 Joules from the power supply.

Mobile Web Browser Optimisations in the Cloud Era It also looks at the benefits of web page prefetching by predicting which links the user will click, and prefetching/loading the relevant data to render the page should the link be clicked more quickly. Does low CPU capability make this a relevant point or not?

The other large aspect they discuss, is the usage of cloud-based architecture. This might increase network latency, and there are other potential security issues, but focusing purely on optimisation, this style of architecture allows processing to be partially taken away from client devices, reducing the workload of the client CPU's. This could be introduced into the chat web app by analysing the impact of moving some intensive functionality from the client to the server.

2.5 Summary

3 Methodology

3.1 Introduction

In order to perform in-depth analysis on the largest areas for optimisation, a web app would need to be studied in depth to gain insights. Rather than study an existing application, over which no control would be granted, a better solution would be to create a testbed application using modern technologies. In tandem with this, Devtools provides accurate and reliable data on browser applications, with the ability to throttle the CPU to judge performance on simulated lower-hardware. This would allow comparisons to be drawn across the different CPU throttle amounts, but with no other parameter changes.

To understand the decision for the case study subjects, first there must be a basic grasp on the history and current usage of JavaScript and ECMAScript.

3.2 A Brief History and Analysis of JavaScript and ECMAScript

JavaScript is a programming language created in 1995. ECMAScript was designed as a way to standardise JavaScript - and was first released in 1997. Versions have been regularly released, with ES6/ES2015 being the current modern standard. For this dissertation, it will be referred to as ES6 for consistency. The issue with ES6 is that not all versions of browsers support ES6, many still only support up to ES5. This means developers are either stuck developing purely in ES5, limiting them to old features, or they can use the Babel compiler to compile their ES6 code into valid and compatibile ES5 - ensuring cross-browser compatibility, whilst having use of ES6 features.

3.3 Testbed Application

The testbed application is a realtime chatting application - an application with a genuine use anywhere in the world. The front-end is built with React, making use of the popular Create React App library, and Socket.io to communicate with a Node.js server also running Socket.io. On top of this, the app makes use of several other libraries, such as Bootstrap, Moment, JQuery and Faker, giving a partially realistic build to the application.

3.4 Case Study One: Effect of Bundlers on Performance

Case study 1 analyses the performance differences between two popular bundlers - Webpack and Rollup. These bundlers are being tested using the testbed

application to ensure consistency across all other parameters other than the bundler itself.

To understand why bundlers are used - it is important to understand the history of JavaScript, and its current state. JavaScript is a programming language, created in 1995 - and has been standardised with ECMA (European Computer Manufacturer's Association). Whilst the 9th edition of ECMAScript (ES2018) was finalised in June 2018, only modern browsers can read ES2015 and above, meaning that if developers want their applications to be comprehensively compatible with browsers, the application must be written in ES2015. This raises the issue that the newer versions of ECMAScript have introduced many features of the language that offer superior development - meaning developers want to write their code in ES2015+. This is where a compiler like Babel comes in. Babel can take ES2015+ code, and convert it into ES5 code.

3.4.1 Bundlers

To adhere to good programming conventions, developers tend to build applications out of many modules, but this raises more issues. To import a JavaScript file in HTML, you use a tag like so:

```
<script src="/path/to/file.js"></script>
<script src="/path/to/file2.js"></script>
etc.
```

If the applications is made up of 200 different files, the browser has to separately send a request for each file, taking a long time to fetch the resources before the user can interact with the full application. This is where bundlers are used. The bundler can take all these files, and bundle them into one JavaScript file - which the browser then requests:

```
<script src="/path/to/bundle.js"></script>
```

Now there is only one file request from the browser, and one file to parse, albeit much larger. Bundlers can do much more than just this however - they can also use plugins like Babel to convert ES2015 to ES5, making the application fully browser compatible, along with minification, setting up development servers and more. This leaves us being able to write modular code, in ES2015+, and still run the optimised app on older browsers. For the purpose of this project, two of the major bundlers were chosen to analyse: Webpack and Rollup.

3.4.1.1 Webpack Webpack is the bundler used by the tool 'create-react-app', popular amongst developers for quickly getting React Apps set up. Webpack turns each module into its own isolated function scope.

3.4.1.2 Rollup Rollup puts everything into the same function scope. When measuring the performance of Rollup - it is anticipated that the execution time should decrease, as its handles modules in a different style to Webpack (Lawson, 2016).

```
function foo(a, b) {
    function bar() {
        return a + b;
    }

    return bar();
}

let result = foo(1, 2);
```

When the JavaScript engine reaches this section of code, it creates a new function object in memory for `foo()`. When the engine reaches the line where `foo()` is called, it executes the function, which creates a new function: `bar()`. When the function execution finishes, the `bar()` object in memory is destroyed. This means if `foo()` is called 100 times, the `bar()` function is created and destroyed 100 times. This costs processing, making it generally slower than non-nested functions.

In each case, the bundler will: * Bundle the JS files * Bundle the CSS files * Compile the JS into ES5 * Set up a server and serve the project

The configuration file for each bundler is in the Appendix.

3.5 Case Study Two: Effect of Different JavaScript Frameworks on Performance

Case study 2 analyses the performance difference between four different

3.5.1 Frameworks

Table 2: Framework Comparison

Framework	Size	Number of GitHub Stars
React	106.6KB	127,752
Preact	3KB	22,385
Inferno	48KB	13,619
Hyperappp	1KB	16,660

React has been chosen as the benchmark framework - as the current leader in industry, and will be compared to three other frameworks, designed as faster alternatives.

3.5.1.1 React "React is a declarative, efficient, and flexible JavaScript library for building user interfaces" (Reactjs.org) It was designed for component-based web development - making it easier for developers to create reusable components for the applications. It uses JSX, which allows usage of HTML tag syntax within components. At the time of writing, React is in use across 735,040 websites (https://www.similartech.com/technologies/react-js). As of January 28th, 2018, React had 1,786,699 downloads, a 152.2% increase from the previous year (https://jsreport.io/javascript-frameworks-by-the-numbers-winter-2018/). https://books.google.co.uk/books?hl=en&lr=&id=NZCKCgAAQBAJ&oi=fnd&pg=PR6&dq=React+js&ots=K
d&sig=GSstOX8Ag0OYJNGM8hT5nNDSlKU#v=onepage&q=React%20js&f=false

3.5.2 Preact

As of 28th January 2018, Preact had 29,385 downloads, a 254.3% increase from the previous year.

3.5.3 Inferno

As of 28th January 2018, Inferno had 29,385 downloads, a 276.5% increase from the previous year.

3.5.4 Hyperapp

Hyperapp is a JavaScript framework for building web applications that uses the Elm-inspired approach to state management. GZipped and compressed, the framework is around 1KB (Bucaran 2018).

3.6 Performance Profiling

To collect accurate metrics about the performance of the chat web app, the Google Chrome Development Tools were used. These tools allow for performance profiling, detailed audits for progressive web apps, and information about load times and page weight. Additionally, they have the capability for CPU throttling, which allows visualising the scale of performance difference between unthrottled and throttled CPU with the same application.

Initially, the unbundled version of the application was analysed, with an unthrottled CPU. This gives the initial set of metrics that can be calculated against. These are:

Table 3: Devtools terminology

Metric	Description
Load time	Total load time for page resources to be rendered
TTI	Time to interactive (e.g When the user can interact with page)
Page Weight	Accumulated total file size
Loading	Parsing, sending/receiving requests
Scripting	Compiling and evaluating scripts
Rendering	Executing page layout
Painting	Drawing the render to the screen
Other	
Idle	Time spent waiting

Then a x4 throttle on the CPU was selected, and the again, giving a new set of results. From here, it was possible to calculate the percentage increase from the unthrottled set to the throttled set.

$$
\begin{aligned}
PI &= Percentage\ Increase \\
T &= Throttled\ CPU\ result \\
U &= Unthrottled\ CPU\ result \\
PI &= \frac{T - U}{U} \times 100
\end{aligned}
$$

Across the results, this gives the full range of percentage increases between the data with no CPU throttling and the application of the 4x throttle. Then the steps were repeated but with a 6x throttle applied, and once again calculated the percentage increase. Across multiple passes, the metrics with the largest percentage increase upon throttling the CPU was rendering, idling and other. Despite these being the largest percentage increases, it must also be taken into account the total time they are taking. A 10% increase of a task taking 1000ms will see a bigger performance hit than a 50% increase of a task taking 5ms. Given the results varied, more passes of the test under the same conditions will be conducted in order to provide the average metric.

To gain the data, Puppeteer was used to run a headless browser, navigate to the URL where the app was being hosted. At this point, the client emulation configuration was decided upon and sent to the headless browser. It could then apply the correct emulation, and begin a timeline trace. Once finished, the trace data would be written to a JSON file, and saved with a name of the current timestamp (to avoid any duplicate file names). The script would run the tracing 20 times each for non-throttled, 4 times throttling, and six times throttling. This reduced outliers, and an average for each metric was collected, before compiling into charts.

Webpack supports all ES5-compliant browsers - and if developers wish to support browsers older than this - polyfills can be loaded to make the application compatible. When using Create React App, webpack's configuration is kept hidden. This default is what will be used in this project to measure performance. This is because the programmer must actively "eject" their application to access customisation of webpack configuration.

Why I chose case 1, 2 etc. (justify with example - expected result) Make a comparison table between predicted/actual results with discussion of relevance Justify reasoning for cases Requirements flow chart + descriptions

4 Design and Implementation

4.1 Case 1 - Bundling

4.1.1 Summary

Whilst the browser can process many unminified JavaScript and CSS files, they can be bundled together into a single file with tools like "Webpack". This means the browser must only load, parse and execute one large file, rather than lots of separate files. "Create React App" uses this by default to optimise applications when they are built for production.

4.1.2 Prediction

Upon bundling the app, there are expectations that the loading time reduces, along with the total page weight. There is anticipation that the rendering and painting times will not fluctuate much if at all - as the same content is still being calculated, and exactly the same page is still being painted onto the screen. Loading and scripting times are expected to be where the most significant increases are seen.

4.1.3 Outcome - Webpack

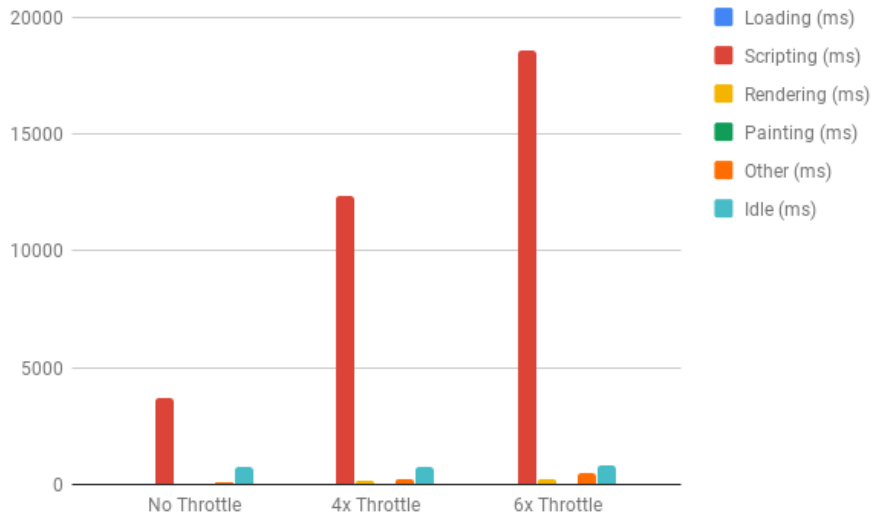


Figure 3: Unbundled Chart

4.1.4 Outcome - Rollup

It is evident from profiling the application that has been bundled with Rollup has significantly reduced scripting times.

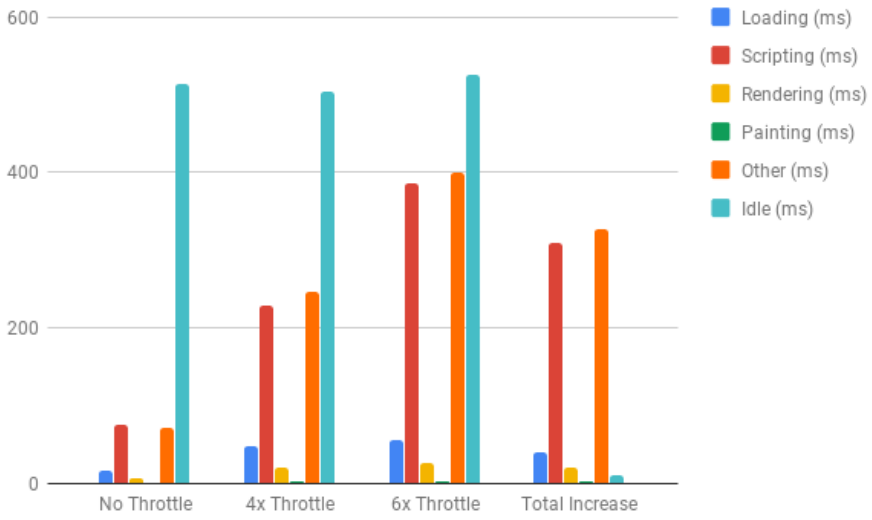


Figure 4: Rollup Chart

4.2 Case 2 - Framework Analysis

4.3 Summary

5 Evaluation and Results

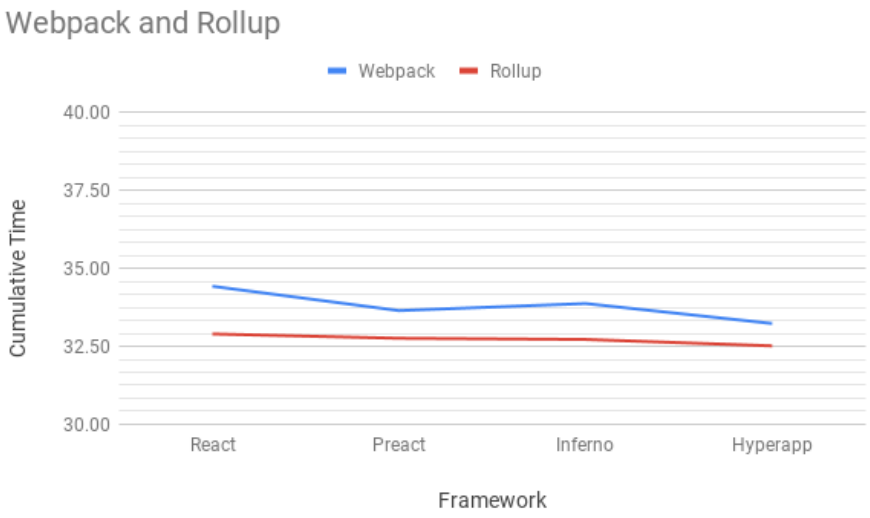


Figure 5: Webpack vs Rollup

Text

6 Discussion

Talk about service workers as a possible improvement on this research. Testing in other browsers, and testing across more machines.

7 Project Management

7.1 Summary and Methodology

A project with multiple complex elements requires careful planning and preparation to be successful, and avoid wasted time in the future. Whilst generally proceedings went smoothly, there were definite lessons learned from this project that can and will be applied to future endeavours. To achieve the final result - the iterative life cycle methodology was employed, allowing the project to get on its feet in the early stages, and gradually improve throughout the entire process. With additions like the alternative frameworks, and the automated data collection script, making use of Puppeteer and the Devtools Web API - this method was effective, particularly for a one-person team. This is where many of the other project management methodologies struggle to have an impact.

7.2 Timeline

Making use of Gantt Chart, and GitHub Projects feature to plan a schedule. Retrospecitvely, the project should have been planned in greater detail, allowing the Gantt chart to be less abstracted - making definitive goal setting more achievable. Alongside this, the supervisor meetings were utilised to draw up goals and criteria to have achieved by the next session.

7.3 Communication

Whilst it took longer than planned to contact the supervisor - a recognised mistake from the author, when meetings became a regular occurance, the project began to build steadily and effectively.

7.3.1 Presentation Feedback

After the presentation, the decision was made to change the focus of case study 2. Whilst partially relevant - the research would have shown that web apps perform better with heavy processing done on a server, leaving the client with a light load, and that a server could send a more optimised bundle for a device. Whilst achievable as an experiment - this research would likely not have shed any new light on the area - therefore the JavaScript frameworks comparison became the new subject for case study 2.

7.4 Legal, Social and Ethical Implications

Potential for degrading software or devices. Any danger of associating devices to parts of the world.

8 Conclusion

Throught the experiments put forward in this dissertation, several points can be identified. Firstly, the choice of bundler and framework do impact the performance of a web page - however, depending on the size of the app, the performance benefits may be negligble.

As researched by Lawson, the choice of bundler becomes more integral upon scaling the usage of modules. When comparing each of the bundlers over multiple throttling thresholds, it was seen that Rollup was in fact demonstrating its marginal benefits - utilising the shared function scope bundling tactic.

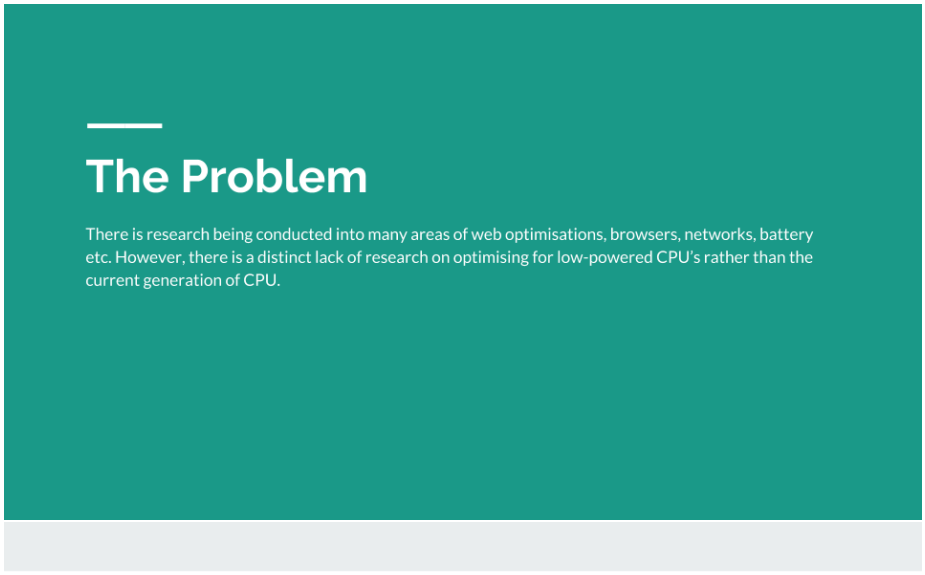
As seen from the full comparison table, the optimal solution when developing for users on lower-power smartphones is the combination of Hyperapp and Rollup. This combination sees the lowest overall score across multiple throttling thresholds.

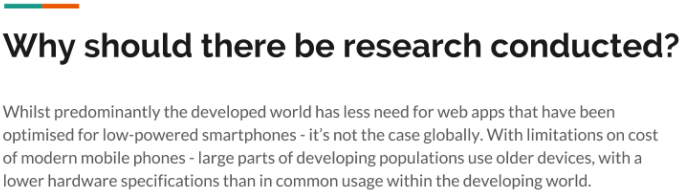
9 Bibliography

- Bucaran, J. (2018) *Hyperapp: The 1 KB Javascript Library For Building Front-End Apps* [online] available from <https://www.sitepoint.com/hyperapp-1-kb-javascript-library/> [24 April 2019]
- Hiscott, R. (2016) *The Cheap Smartphones Competing For The Developing World's Internet* [online] available from <https://www.dailydot.com/debug/cheap-smartphones-for-the-developing-world/> [22 April 2019]
- Lawson, N. (2016) *The Cost Of Small Modules* [online] available from <https://nolanlawson.com/2016/08/15/the-cost-of-small-modules/> [17 April 2019]
- Sharwood, S. (2017) *Developing World Hits 98.7 Per Cent Mobile Phone Adoption* [online] available from <https://www.theregister.co.uk/2017/08/03/itu_facts_and_figures_2017/> [22 April 2019]

10 Appendices

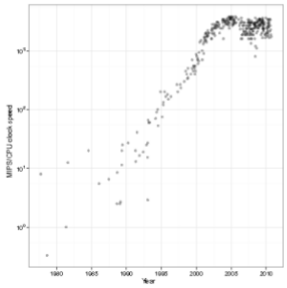
10.1 Presentation





Moore's Law

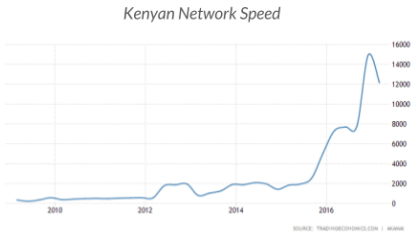
Over the last 10 years, Moore's Law has begun to plateau - and suggests at least in the near future, the little increase in computational power may continue to preside.

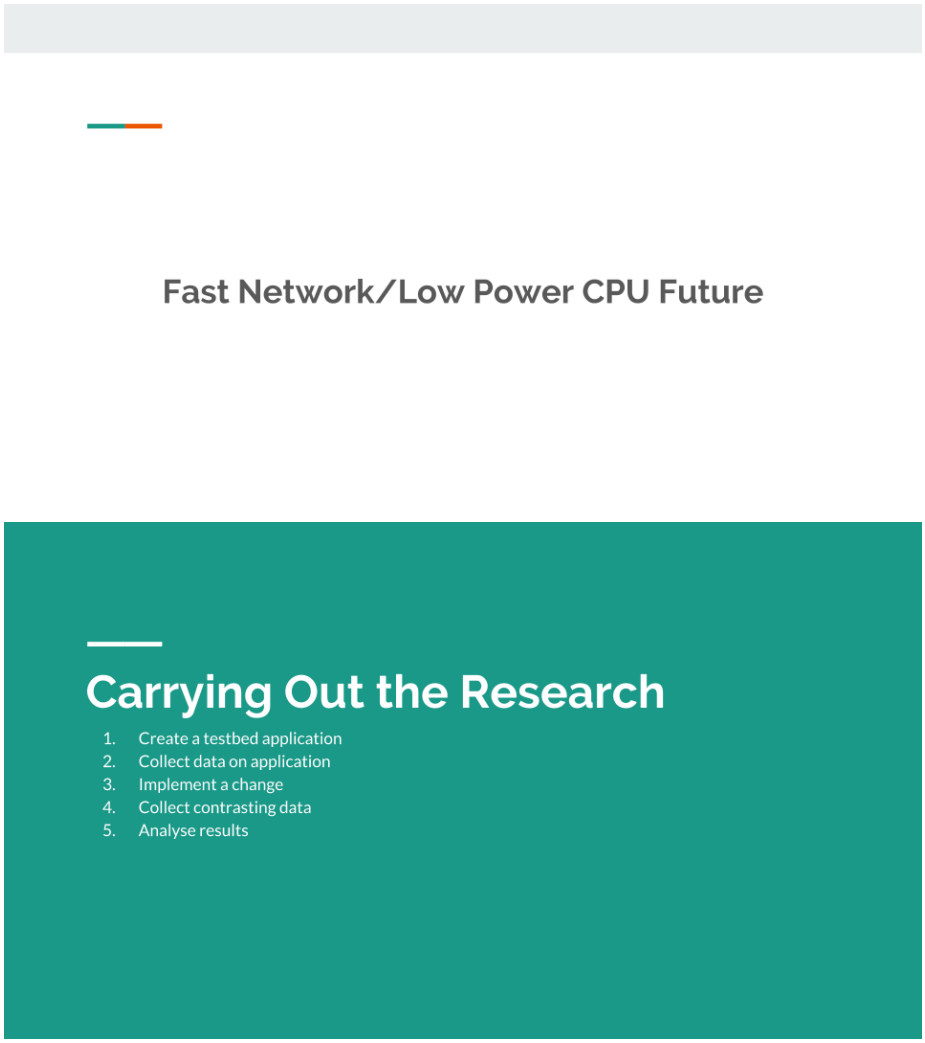


Network Speeds

In contrast to this - there has been a general shift in network speeds increase over the last 10 years in Kenya.

Should these trends continue - this would leave a fast-network/low-powered CPU future.




Fast Network/Low Power CPU Future
Carrying Out the Research
1. Create a testbed application
2. Collect data on application
3. Implement a change
4. Collect contrasting data
5. Analyse results

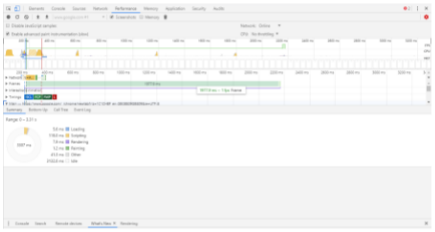
Testbed Application

A testbed realtime chatting application - utilising modern industry-standard technologies (React, Node.js, Express.js, Socket.io) was created with which to experiment - granting the ability to gather in-depth metrics upon applying new technologies (webpack, rollup, parcel, server-decision tools).

Google Chrome Devtools

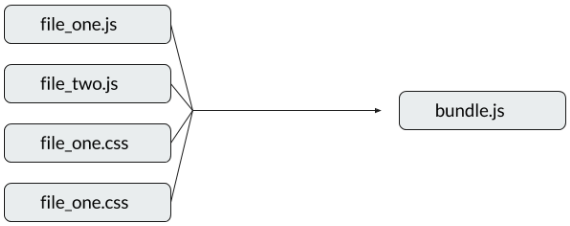
The Google Chrome Devtools is the industry standard set of tools that developers use in the browser to gain insights into how code is interacting with the browser.

There are also tools like Puppeteer to allow headless browser automation, giving easier access to more results within a given timeframe.



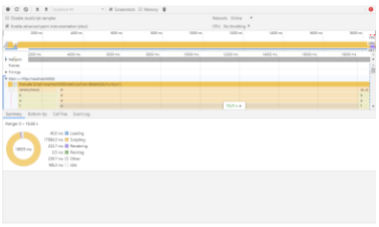
Case 1: Bundling an Application

What is bundling?



Webpack

Data Collection with Chrome Devtools

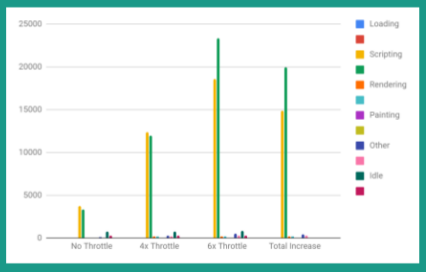


Running the app in its current state through the Devtools performance profiler yields useful data about the load times and performance metrics.

This data can then be scraped and put into a spreadsheet for further analysis.

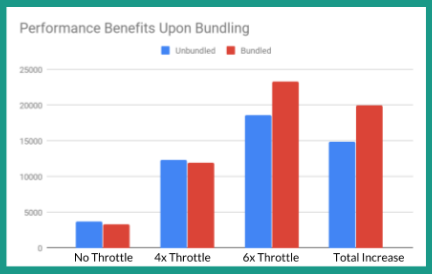
Using a custom script - much of this process could be automated for faster collection - meaning more results could be collected.

Results

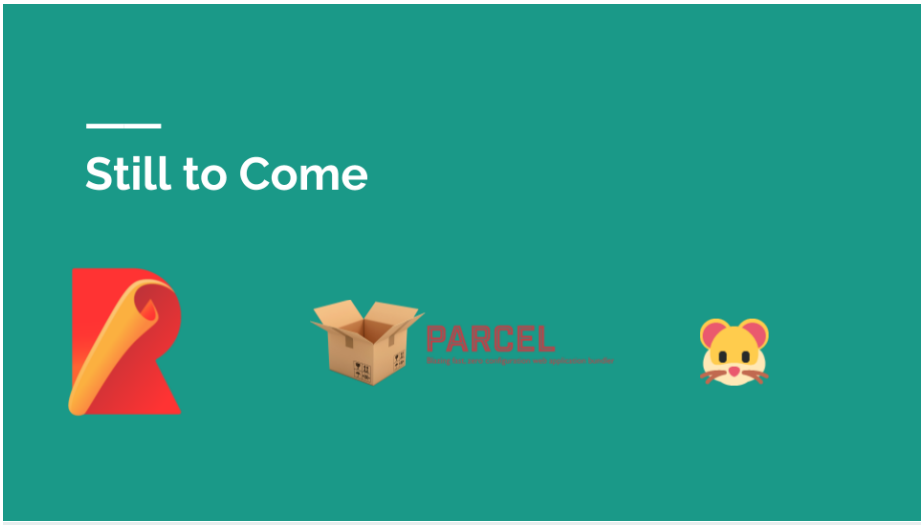


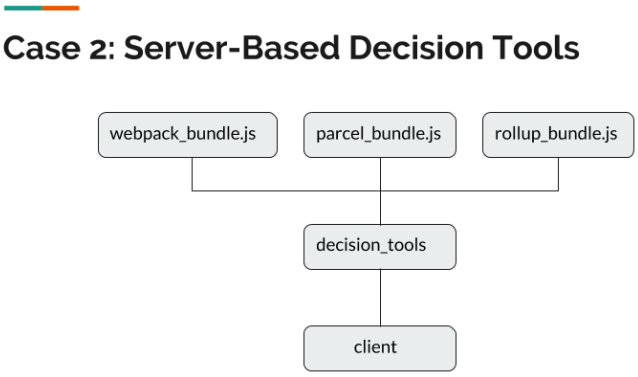
Compiling the results into a spreadsheet, and consequently into a chart showed that compared to processes that occurred in the scripting heading, loading, rendering, painting, idling, and other task performance was negligible. This gave a key area of optimisation to focus on.

Scripting in isolation



Whilst at no throttle or a 4x throttle, the bundled version does perform marginally more efficiently - at a 6x throttle, the bundled version takes a large hit, leaving the total increase from no throttle to 6x throttle a large difference.


Still to Come
PARCEL


Case 2: Server-Based Decision Tools
webpack_bundle.js
parcel_bundle.js
rollup_bundle.js
decision_tools
client

Client vs Server-Based Architecture

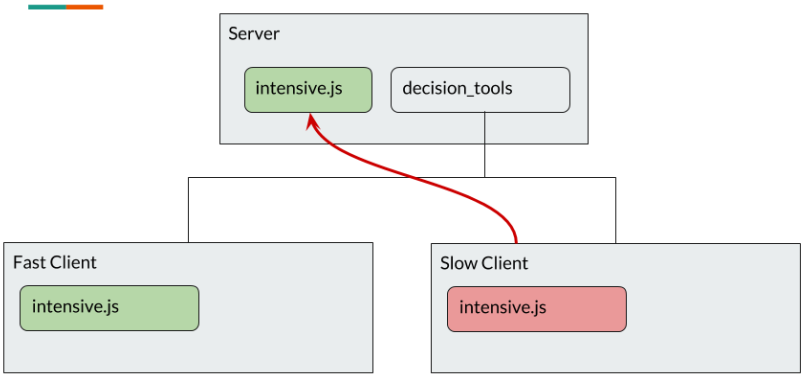
Web technology has seen shifts in desires for architecture to be client and server based. With powerful modern devices - it is becoming more common for the heavy lifting to be done on the client - leaving the server running lightly to keep high speeds with high volumes of traffic.

With low-powered devices, if developers wish for fast user experiences, then the server-based architecture become relevant again.

Accessing the Client

JavaScript is now capable of acquiring an estimate of the processor speed that is executing the script. This means an application can tell the server basic information about the device - allowing the server to make useful decisions on what it should and shouldn't send/process.

This optimises processes for specific devices, keeping servers quick and using client-based architecture where devices are capable - and switching to server-based architecture and optimised bundles for older browsers and slower architecture.



TODO

- ~~Create testbed application~~
- ~~Decide on data collection strategy~~
- ~~Implement automated solution~~
- ~~Collect data on webpack bundling~~
- ~~Analyse results and produce charts~~
- Gather metrics for other bundlers and compare results
- Implement server-side decision tools and analyse usage results
- Complete write up and formalities (statement of originality etc.)

Bibliography

- Gough, N. (2005) *Africa: The Impact of Mobile Phones* [online] available at <https://www.share4dev.info/telecentreskb/documents/4552.pdf> [26 January 2019]
- Iliev, I. (2014) *Front end optimization methods and their effect* [online] available from <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6859613> [03 February 2019]
- ITWeb (2018) *Africa Sees Strong Internet Growth* [online] available from <https://www.itweb.co.za/content/VgZeyvJA3V6qdjX9> [11 February 2019]
- Simonite, T. (2016) *Intel Puts the Brakes on Moore's Law* [online] available from <https://www.technologyreview.com/s/601102/intel-puts-the-brakes-on-moores-law/>
- [03 February 2019]
- Trading Economics (2017) *Kenya Internet Speed* [online] available from <https://tradingeconomics.com/kenya/internet-speed>[10 February 2019]
- Wang, H., Kong, J., Guo, Y., Chen, X. (2013) *Mobile Web Browser Optimizations in the Cloud Era: A Survey* [online] available from <https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6525571> [10 February 2019]

10.2 Unbundled Profiling

10.2.1 At No Throttle

Table 4: Unbundled at No Throttle

Run (0x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
1	4.7	3111.8	43.4	0.6	60.9	453.4
2	4.1	3431.6	48.2	0.7	62.4	5100
3	4.8	3180.1	47.4	0.9	65.6	440.6
4	1	3197.6	50.7	0.4	36.1	156.2
5	4.7	3899.6	77.9	0.7	72.8	539.3
6	7	3834.2	54.9	1.3	88.1	477.8
7	4.9	3941.5	62.4	12.8	83.9	617.5
8	5.3	3694.9	53.2	1.4	94.1	515.8
9	5.2	4164.4	59.1	0.7	84.2	670.3
10	5.7	3434.5	59.3	3	157	804.3
11	4.6	3581.2	55.8	2.6	77.2	682.4
12	4.2	3206.8	45.5	0.6	71.9	524.1
13	5.8	4151.7	53.5	0.7	88.5	500.6
14	4.1	4656.3	59.2	0.8	78.9	555.5
15	12	4185.6	57.1	1.2	80.9	664.8
16	4.2	3698.1	51.7	1.1	84.2	492.3
17	4.3	3724.6	49.5	2.6	78.6	496.5
18	5.4	3618.6	53.3	0.9	83.2	496.6
19	5.4	4018.2	57.1	1	85.8	525

Run (0x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
20	4.6	3551.9	50.9	0.7	82.7	490.1
Average	5.1	3714.16	54.505	1.735	80.85	760.155

10.2.2 At 4x Throttle

Table 5: Unbundled at 4x Throttle

Run (4x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
1	15.5	12827.6	164.2	2.1	186.8	645.7
2	6.2	10566.3	159.9	1.9	214.1	734.5
3	12.9	13499	155.4	2.1	205.1	671
4	19.8	12701.8	155.1	2.2	193.6	898
5	10.7	9852.5	187.1	6.7	372	713.9
6	11	12833.3	178.4	7.9	261.2	947.5
7	7.2	11398	180.7	6.7	290.4	661.1
8	16.1	12382.4	165.8	2.4	192.2	799.7
9	12.1	12313.2	181.9	2.3	313.2	932.5
10	15.8	14077.1	180.6	2.1	274.5	750.2
11	18.3	7225.1	169.9	2.3	190.4	774.6
12	9.3	13893.1	187.2	2.4	121.5	594.6
13	13.2	13131.3	199.9	5.2	234.9	789.8
14	9.8	14363.4	167.2	1.7	199.9	967.2
15	19	11134.7	168.8	6.6	219	553
16	19.5	12194.9	177.9	3.7	235	627.1
17	11.8	12514.6	187	2.8	186.3	942.6
18	14.6	12724.4	160.5	3	282.4	797.2
19	26.1	15305.5	172.1	3.2	398	816.5
20	16.3	12208.5	176.5	6.3	242.7	584.7
Average	14.26	12357.335	173.805	3.68	240.66	760.07

10.2.3 At 6x Throttle

Table 6: Unbundled at 6x Throttle

Run (6x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
1	8.5	19643.1	211.8	2.4	246.7	577.7

Run (6x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
2	6.4	14932.7	191.6	2.6	644.9	631.9
3	23.7	19366.7	254.1	3.3	244.1	552.2
4	18.3	19754.3	184.8	4.3	351.3	538
5	11.4	19008.9	227.9	2.9	715.7	668.5
6	25.9	21328.3	108.5	14.9	1622.9	1117.2
7	9.6	14051.8	239.6	3.1	365	1273.9
8	9.2	19393.9	188.9	4.9	334.8	715.2
9	17.5	21682.2	271.2	4.1	529.6	980.9
10	21.3	22173.7	243.6	4	835.8	851.5
11	9.2	17532.4	256	4.4	320.8	921.1
12	8.2	19373.5	251.3	2.7	213.5	654.6
13	8	17301	224.3	1.8	387.6	1423.8
14	11.2	20298.6	118	2.9	512.2	1036.1
15	11.5	16325	232.9	10.1	220.3	797.8
16	9.8	18459.1	178.3	1.4	303.8	684.5
17	22.3	18405.5	380.6	3.9	249.3	852.9
18	28.6	19070.7	224.5	1.9	1249.2	562.1
19	15.2	13584.4	177.1	1.1	307.5	794.6
20	10.1	20168.3	260	3.1	319	662.9
Average	14.295	18592.705	221.25	3.99	498.7	814.87

10.3 React Webpack

10.3.1 At No Throttle

Table 7: Bundled at No Throttle

Run (0x)	**Loading (ms)**	**Scripting (ms)**	**Rendering (ms)**	**Painting (ms)**	**Other (ms)**	**Idle (ms)**
1	12.2	2852.9	54.7	1.3	45.9	524.6
2	13.1	3011.8	67.6	1	41.6	466
3	13.4	2916.5	52.7	1	43.3	465.9
4	12.1	2790	51.6	0.7	41.5	479
5	13.5	2952.6	54.9	0.8	43.3	500.5

Run (0x)	**Loading (ms)**	**Scripting (ms)**	**Rendering (ms)**	**Painting (ms)**	**Other (ms)**	**Idle (ms)**
6	13.6	2769.9	55.6	0.8	41.6	558.1
7	13.8	2935.2	53.6	0.9	57.3	454.2
8	11.2	2869.7	56.3	0.8	50.4	520.7
9	13.2	3969.4	51.3	0.6	48.9	568.5
10	13.5	2838.4	52.9	0.8	45.2	436.2
11	13	2867	54.7	0.8	42.7	454.7
12	12.8	2826.9	50	0.6	46	492.1
13	12.1	2902.6	50.5	0.9	45.8	490.6
14	12.5	2929.5	50.7	1	46.1	469.7
15	12.2	2960.9	58.4	0.7	43.5	573.8
16	13.2	2844.5	50.3	1.2	55.6	476.6
17	11.9	2934.4	51.4	0.7	42.5	475.6
18	12.5	2965.1	51.5	0.7	43.8	450.5
19	13.2	2968.4	52.2	1.4	48.4	438.2
20	12.7	2930.3	52.6	1	45.1	471.6
Average	12.785	2951.85	53.675	0.885	45.925	488.355

10.3.2 At 4x Throttle

Table 8: Bundled at 4x Throttle

Run (4x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
1	48.3	12215.3	162.9	0.8	94.3	111.3
2	55.7	13474.3	172.7	5.5	180.6	302.2

Run (4x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
3	33.9	12209.6	167.8	5.9	211.7	260.7
4	32.9	11922.5	160.7	0.6	251.3	378.4
5	33	10846.5	172.4	0.5	243.4	434.5
6	31.5	11881.3	175	4.5	115.2	136.6
7	62.7	12602.4	174.8	0.9	132.6	144.9
8	64.3	12968.5	176.2	0.9	235	207.6
9	49	7734.3	168.8	26.3	253.6	183.1
10	23.6	11292.8	161	0.3	215.6	481.1
11	38.4	13622.1	171.4	1.3	161.5	367
12	16.8	11135.7	153.9	0.3	229.3	366.1
13	28.7	10936.2	161	0.7	292.6	306
14	32.8	10993.3	177	2	100.9	201.8
15	60.8	11098.9	176.4	2	205	194.9
16	26.9	13087.2	157.7	9.3	246.4	154.1
17	31	12473.6	162.4	1.9	179.5	829.6
18	26.6	12748	167.7	2.3	235.9	241.2
19	26.6	12748	167.7	2.3	235.9	241.2
20	34.3	13010.6	170.5	7.6	118.2	209.2
Average	37.89	11950.055	167.9	3.795	196.925	287.575

10.3.3 At 6x Throttle

Table 9: Bundled at 6x Throttle

Run (6x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
1	40	17984.3	222.7	0.5	239.7	168.3
2	36.4	18407.9	380.5	17	191	396.8
3	74.2	19338.6	232	1.3	151.7	365.2
4	65.3	20081.5	157.3	1.9	378.3	591.8
5	30.6	121064.4	237.4	5.1	185.7	276.7
6	44.6	17446.2	215.9	0.9	165.2	210
7	31.8	17675.2	241.1	1.1	237.8	236.4
8	68.5	19803.6	289.5	22.9	234.3	266.5
9	34.1	13028.6	231.6	0.9	93.4	490.3
10	25.9	18598	204.4	4.9	153.2	186.2
11	37.3	17001.4	245.4	0.3	483.1	306
12	38.6	17856.6	219.2	0.8	255.2	120.8
13	22.4	17026.3	214	0.4	472.8	411.7
14	35.7	20385.4	213.8	26.7	341.1	264.9
15	45.9	17995.6	235.1	2.6	314.4	221

Run (6x)	Loading (ms)	Scripting (ms)	Rendering (ms)	Painting (ms)	Other (ms)	Idle (ms)
16	30.7	18087.9	266.2	1.4	182.5	288.3
17	93.3	19167.8	219.5	0.8	249.1	366
18	107.2	19614.1	273.9	7	748.3	189.3
19	104.1	19798	223.4	20.7	501	251.4
20	21.3	16025.1	222.7	1	345.4	519.9
Average	49.395	23319.325	237.28	5.91	296.16	306.375