Bachelor of Science in Software Engineering
June 2020

Performance of Multi-threaded Web Applications using Web Workers in Client-side JavaScript

Johan Djärv Karltorp
Eric Skoglund

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Bachelor of Science in Software Engineering. The thesis is equivalent to 10 weeks of full time studies.

Contact Information:
Author(s):
Johan Djärv Karltorp
E-mail: jokl17@student.bth.se

Eric Skoglund
E-mail: ersk17@student.bth.se

University advisor:
Emil Folino
Department of Computer Science

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context - Software applications on the web are more commonly used nowadays than before. As a result of this, the performance needed to run the applications is increasing. One method to increase performance is writing multi-threaded code using Web Workers in JavaScript.
Objectives - We will investigate how using Web Workers can increase responsiveness, raw computational power and decrease load time. Additionally, we will conduct a survey that targets software developers to find out their opinions about performance in web applications, multi-threading and more specifically Web Workers.
Realization (Method) - We created three experiments that concentrate on the areas mentioned above. The experiments are hosted on a web server inside an isolated Docker container to eliminate external factors as much as possible. To complement the experiments we sent out a survey to collect information of developers' opinions about Web Workers. The criteria for the selection of developers were some JavaScript experience. The survey contained questions about their opinions on switching to a multi-threaded workflow on the web. Do they experience performance issues in today's web applications? Could Web Workers be useful in their projects?
Results - Responsiveness shifted from freezing the website to perfect responsiveness when using Web Workers. Raw computational power increased at best 67% when using eight workers with tasks that took between 100 milliseconds and 15 seconds. Over 15 seconds, sixteen workers improved the computational power further with around 3% - 9% compared to eight workers. At best the completion time decreased with 74% in Firefox and 72% in Chrome. Using Web Workers to help with load time gave a big improvement but is somewhat restricted to specific use cases.
Conclusions - Using Web Workers to increase responsiveness made an immense difference when moving tasks that is affecting the users responsiveness to background threads. Completion time for big computational tasks was quicker in use cases where you can split the workload to separate portions and use multiple threads in parallel to complete the tasks. Load time can be improved with Web Workers by completing some tasks after the page is done loading, instead of waiting for all tasks to complete and then load the page. The survey indicated that many have performance in mind and would consider writing code in a multi-threaded way. The knowledge about multi-threading and Web Workers was low. Although, most of the participants believe that Web Workers would be useful in their current and future projects, and are worth the effort to implement.

Keywords: Web Workers, Multi-threading, JavaScript, Web browsers

Nomenclature

Central Processing Unit (CPU) - The CPU, also known as a central processor, microprocessor or chip, is the unit that executes instructions to make up a computer program [1].

Single-core - A single-core microcontroller has just one processor inside [2].

Multi-core - A multicore microcontroller has two or more processors, also called cores, inside one chip [2].

Operating System (OS) - An OS is the software that manages computer hardware and software resources allowing a user to run other applications on a computing device [1].

Web Browser - A web browser takes you anywhere on the internet. It retrieves information from other parts of the web and displays it on your desktop or mobile device. The information is transferred using the Hypertext Transfer Protocol, which defines how text, images and video are transmitted on the web. This information needs to be shared and displayed in a consistent format so that people using any browser, anywhere in the world can see the information [3]. Some examples of web browsers are Google Chrome, Mozilla Firefox, Safari and Internet Explorer.

Load distribution - In computing, load distribution or load balancing is the process of redistributing the work load among nodes of the distributed system to improve both resource utilization and job response time while also avoiding a situation where some nodes are heavily loaded while others are idle or doing little work [4].

Client-side script - Client-side scripting requires browsers to run the scripts on the client machine and does not interact with the server while processing the client-side scripts. Example of programming languages that is client-side scripts are HTML, CSS and JavaScript [5].

Parallel calculations - Parallel computing is a type of computation in which many calculations or the execution of processes are carried out simultaneously [6].

Application Programming Interfaces (API) - APIs are constructs made available in programming languages to allow developers to create complex functionality more easily. They abstract more complex code away from you, providing some easier syntax to use in its place [7].

Web APIs - Web APIs are built into your web browser and are able to expose data from the browser and surrounding computer environment and do useful complex things with it [7].

Document Object Model (DOM) - The Document Object Model (DOM) is a programming interface for HTML and XML documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects. That way, programming languages can connect to the page [8].

Container - A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another [9].

Docker - A platform for containerized applications. Docker provides the ability to package and run an application in a loosely isolated environment called a container. The isolation and security allow you to run many containers simultaneously on a given host. Containers are lightweight because they don't need the extra load of a Hypervisor, but run directly within the host machine's kernel [10].

Effectiveness - According to ISTQB, the definition of effectiveness is the extent to which correct and complete goals are achieved as quickly as possible [11].

Responsiveness - The task of a system or function to complete assigned tasks within a reasonable given time frame. The user should be given feedback frequently to avoid the user believing the system is not functioning as expected. The user experience should feel smooth and feedback from the system should be instantaneously. To measure responsiveness frames per seconds is often used. Frames per second is how many times the pixels on the screen is updated per second. This is according to our definition used in this theses.

Load time - The point at which a program is put into an executable state and ready to interact with the user; literally, the time at which it is loaded. This is according to our definition used in this theses.

Raw computational power - The amount of useful work accomplished by a computer system. Computational power is estimated in terms of executing computer program instructions and tasks as efficiency and quickly as possible. This is according to our definition used in this theses.

Contents

Chapter 1

Introduction

1.1 Background

JavaScript is a common programming language used for web development and many web applications are built with it. JavaScript was created to allow client-side script to be executed to create dynamic websites that could interact with the user in real time. JavaScript was developed by Brendan Eich for Netscape and had its first appearance in 1995 [12]. JavaScript is useful for communication on the web, and has the ability to accomplish things, such as; managing the browser, editing content on a document (DOM), and let client-side scripts communicate with users in asynchronous communication [13]. One thing to know about JavaScript which runs web applications is that it is limited to only use the main thread inside a web browser. That means only one command at the time can be executed. If you are running multiple actions at the same time, that makes the browser freeze and wait for the previous command to be done, which makes it impossible for the user to interact with the web browser during that process.

Lately, CPUs are becoming more and more multi-core dependent, in order to increase performance [14]. This means that single-core performance has stagnated over the years. From 1995 to 2004 single-threaded performance improved with 52% per year and from 2004 to 2011 it decreased to only 21% improvement per year. Without Intel CPU's which have always focused heavily on improving single-threaded performance it dropped further to 13% per year [15]. To attain full advantage of performance on the CPU, multi-core workflows are required. However, web applications are still limited to single-threaded workflows.

So how does JavaScript handle multiple command executions?
JavaScript uses a stack, callback queue, and something called the ***Event loop***. The main purpose with the event loop is to make sure the requested commands are being executed. When a function is called to be executed it is put on the stack. The stack handles all the requests and runs them in the order they arrive. An example is to run a function with delay. The function is processed in the web API in the browser where it waits for the timeout to be completed. Once the timeout is done the function is sent to the task queue. That is where the event loop comes into play. The event loop notices that a task is waiting in the queue. In order for that task to be executed, the event loop has to make sure the stack is empty. If it is empty the task can run, otherwise it has to wait. An example is illustrated in figure 1.1 [16].

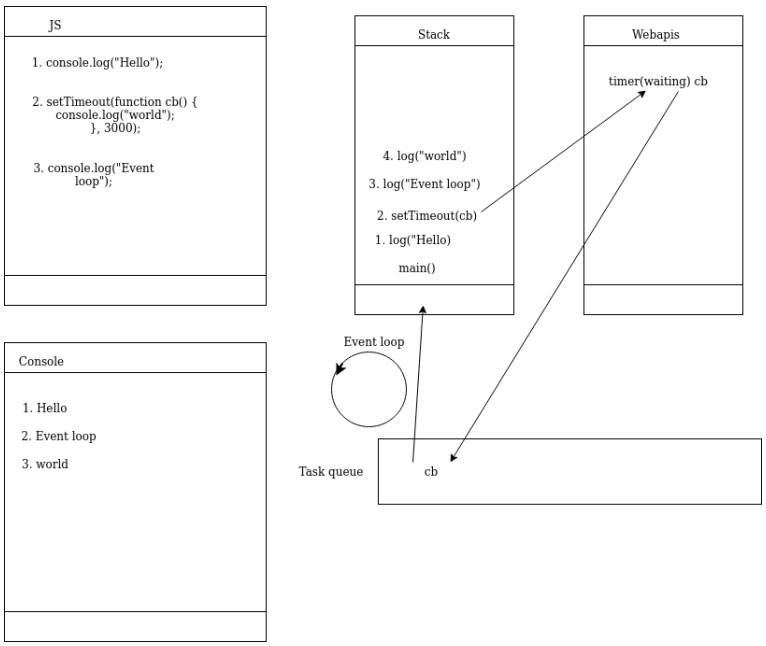

Figure 1.1: An illustration on how event loops operate.

We know that JavaScript is limited to a single-threaded workflow, which means every new process has to wait for the running process to be completed. If the application runs multiple processes at the same time, that may lead to long waiting time. The term for handling this situation is ***multi-threading***. Multi-threading allows multiple threads to run in the same process. What happens is that threads take turns running on the CPU. By switching between the different threads, the system provides an illusion of threads running in parallel on a slower CPU than the actual CPU in your computer. If one process contains four compute-bound threads, the threads would appear to be running in parallel on the CPU, with one-fourth the speed of the actual CPU in your computer. Since threads have the same address space, they share variables, memory addresses, open files, child processes, and so on. This means threads can cooperate rather than fight about the resources. An example of a process containing multiple threads is illustrated in figure 1.2 [1].

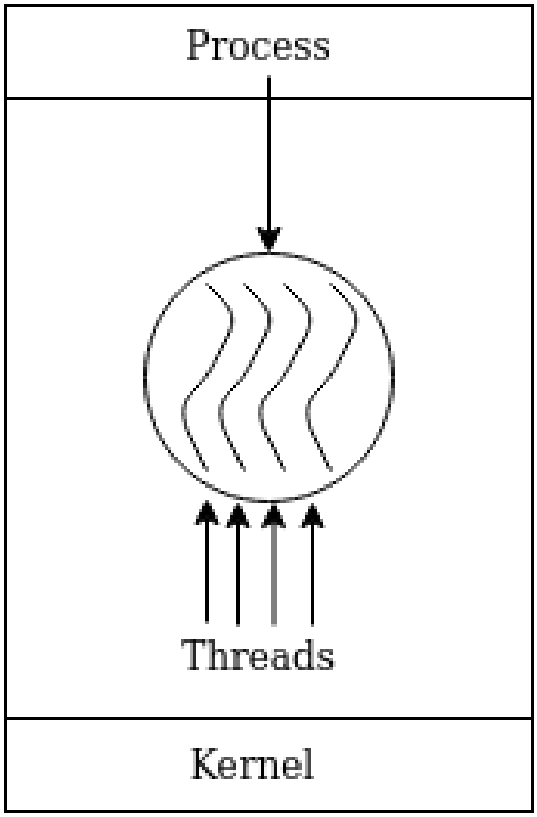

Figure 1.2: One process with four threads.

Many programming languages e.g. C & C++ (POSIX Threads) [17], Java [18], GO [19], Python [20] have been supporting multi-threaded solutions for a long time. JavaScript has not been one of them, until a couple of years ago. A new API called ***Web Workers*** was released, which made it possible to run JavaScript code multi-threaded.

1.2 Purpose

In the past decade, software applications on the web have significantly increased. Traditionally desktop applications such as word processors, spreadsheets, calendar, games, instant messaging etc. can now be accessed on the web. The reasons behind the web becoming a popular deployment environment is; that it is easily accessible around the world, it does not require download and installation, automatic updates to users and facilitates collaboration among users [21]. This also means that more demanding applications is running in your browser.

A recent web API ***Web Workers*** has been developed that facilitates for web content to run scripts in background threads. These background threads can execute code without interfering with the main thread where the user interface is handled [22]. This method may significantly increase responsiveness and performance for web applications that are performing heavy tasks which could benefit from more cores. We want to explore how Web Workers operate and perform experiments on how much load time, responsiveness and raw computational power differs when using Web Workers. Is the implementation cost worth the performance gain?

Our other purpose with the research is to find out developers' opinions about Web Workers. It does not matter if Web Workers are amazing if nobody uses them. We created a survey to find out what the community thinks about Web Workers.

1.3 Scope

JavaScript

The first thing that will be covered in this thesis is JavaScript. The focus will be on client-side JavaScript. The text includes a description of JavaScript itself, how it works as a single-threaded language, and what the event loop is. It does not include server-side JavaScript.

Multi-threading

The second area that will be covered is multi-threading. This includes a basic description of what multi-threading is. How parallelization works, and what separation of threads is. It does not include issues that may occur with multi-threading, such as process synchronization and race conditions.

Web Workers

The third area that will be covered is Web Workers. This is the foundation of the whole thesis. This text includes an overview of what a Web Worker is. A list of the different kinds of workers, along with a brief description of them. The term thread safety will be explained. An example of Web Workers is provided.
Furthermore, what does Web Workers improve in terms of performance, such as responsiveness, heavy computations, and load time. How does parallelism solve issues with long running tasks and does it handle tasks that were not feasible before.
There are not only benefits with Web Workers. The downsides will be covered as well. That includes high start-up performance cost, access issues, and the complexity they entail. The focus will be on Dedicated Workers, not Shared Workers nor Service Workers. The final thing is the performance results in different web browsers, in this case Chrome and Firefox. The experiments were executed in these web browsers, not all browsers. The reason why we chose those web browsers is because of its popularity [23].

JavaScript developers' community

The last area that will be included in this scope is opinions about Web Workers from developers. We have focused on asking developers with JavaScript experience, not inexperienced developers.

Chapter 2

Research Questions

2.1 RQ1

How much does performance differ between Firefox and Chrome when using Web Workers?

We are interested to find out if results may differ depending on which web browser you are using. How will the performance results differ when letting Web Workers run the same experiments in different web browsers?
The goal with this question is to distinguish which browser between Firefox and Chrome who handles load time, responsiveness, and raw computational power best.

The outcome will determine which browser delivers the greatest performance results and how much it differs. The expected outcome is that the difference in the results will be minimal between the web browsers. The conclusion of this experiment will be to see which browser has better performance and would be more suitable to use when working with Web Workers.

2.2 RQ2

What is the threshold of effectiveness for Web Workers?

The goal with this question is to find out where the threshold of effectiveness for Web Workers lies, in order to determine at which point it is worth using them. The results will indicate when Web Workers are considered useful and when not. The expected outcome of this experiment is that Web Workers will only be considered useful when processing huge amounts of data. But how much data that is required is unknown. The conclusion of the results will be to see how much data is needed to be processed to determine when Web Workers operate best.

2.3 RQ3

What are developers' opinions about Web Workers?

To answer this question we are interested to get software developers opinions in the subject. The goal is to retrieve information from developers about their knowledge in Web Workers and multi-threaded web applications. Since Web Workers are pretty new in JavaScript we do not expect that many acquire much knowledge about them, hence, even less experience working with them. Since we expect the amount of data required for Web Workers to be efficient to be high, we do not expect many developers would consider the effort of implementing them. The conclusion of this will be to obtain other developers' opinions about Web Workers to find out if the potential performance gains would be considered worth the effort of implementing them.

Chapter 3

Research Method

RQ1 and RQ2 will be answered by running different experiments focusing on responsiveness, load time, and computational power, to see where the threshold of effectiveness occurs for Web Workers. The results from the experiments will determine which use cases where Web Workers operates best.

We will be using Docker to host our web server. This means our server runs inside an isolated container and external influences such as the operating system, network and concurrent processes should not affect the result. This gives our experiments a more stable foundation to stand on, which in return will give us similar results on each run. This also makes replicating our study much easier and confirming our results.

At the same time we will be collecting performance results from the web browser that is running the experiments. Here lies the answers to RQ1. By measuring completion time for each experiments from both Google Chrome and Mozilla Firefox, and later be compared against each other. Again the reason why we chose those web browsers is because of its popularity [23].
We will analyze the performance results to find potential bottlenecks for both web browsers when working with Web Workers.

RQ3 will be answered by collecting and analyzing the information coming from the survey. The type of questions that will be covered are; interest in multi-threaded applications on the web, knowledge about Web Workers, performance issues inside web applications, have Web Workers improved performance results, potential issues with Web Workers and could Web Workers be useful for current or/and future projects.

3.1 Experiments

Software versions and hardware configuration used in the experiments can be found in chapter 9 - Supplemental Information.

3.1.1 Computational power

In this experiment we are testing the computational power when using Web Workers. We are calculating prime numbers with 0, 1, 2, 4, 8 and 16 workers. The experiments were executed ten times in each web browser and the result is based on the average value from all these results. The result should show us when zero, few or many workers should be used to increase performance and when workers start to give diminished improvements. Our implementation for calculating prime numbers is not the most effective way, but in our case that does not matter because we are using the same implementation on all the tests. Why we chose to specifically calculate prime numbers to measure raw computational power, was the few lines of code needed to implement and the ease to increase data size against different numbers of workers. To check if the number is a prime number this function is used:

```
const isPrime = num => {
    for (let i = 2; i < num; i++)
        if (num % i === 0) return false;
    return num > 1;
}
```

To take full advantage of multiple threads, each worker is given an unique chunk of the total number of primes to calculate by using this function:

```
let chunk = size / nrOfWorkers;

for (let i = 0; i < nrOfWorkers; i++) {
    workers = new Worker("worker.js");
    workers.onmessage = workerDone;
    workers.postMessage({ start: chunk * i, end: chunk * (i + 1) });
    running++;
}
```

This means every worker will calculate different prime numbers at the same time to achieve parallel calculations. Each chunk is then sent back and added to the result. When all workers have returned their finished chunk the timer stops.

We measured the completion time for 10,000, 20,000, 40,000, 80,000, 160,000, 320,000, 640,000 and 1,280,000 prime numbers. In our graphs we included the different completion times for Firefox and Chrome.

3.1.2 Responsiveness

To test responsiveness we will create a slow and simple spell checking program that needs to be executed every time the user types a new word and hits the space-bar. The word is then evaluated and compared with the English dictionary to determine if it is a correct word or not. This experiments was executed ten times in each web browser. Why we chose to measure responsiveness by using user input was because that is when a user would acknowledge and experience the poor responsiveness the most. When a user is typing, the feedback from the program needs to be instantaneous. Under 0.1 second is desired to make the user feel like they are directly manipulating the text freely [24]. This gives us a good threshold to strive towards, and when responsiveness starts to decrease a clear and instance change will be noticed by the user typing and in the measuring tools. We will first test to run the spell checking on the main thread and then run the spell checking on a Web Worker thread instead.

To measure responsiveness inside Firefox we used **Firefox Developer Tools - Performance** [25]. The tool records the frames per seconds during the session. This makes it very easy to see if responsiveness is effected by big drops in the graph.

To measure responsiveness inside Chrome we used **Chrome DevTools - Runtime performance** [26]. This tool displays responsiveness a bit differently compared to Firefox. Instead of drops in the graph, a red bar is displayed if the frame rate drops so low that it is harming the user experience. The threshold when the red bar appears is determined by the tool itself and no clear documentation is found of what the threshold is based on.

3.1.3 Load time

To test load time, the experiment will focus on long running jobs that need to complete before the web site can load and to see if Web Workers can help to take some of the workload off the main thread. This experiment was executed ten times in each web browser. The final result in based on the average value of these runs.

The website is simply trying to fetch a text file and display it for the user. In our experiment we are fetching the bible text. After the script has fetched the text it starts to run an intensive workload. The intensive workload could be anything from fetching resources (images, files, queries, ...), applying styling, computational functions or anything that is not critical to complete before the page loads.
Our intensive workload looks like this:

```
let intensiveWorkLoad = () => {
    let i;
    for (i = 0; i < 8000000000; i++) {}

    p = document.createElement("p");
    p.innerHTML = "done\n";

    return i;
}
```

Loading the text and the intensive workload is called directly when the script is loaded. Because of the way JavaScript files are loaded by the web browser the intensive workload function needs to complete before the website can continue to load the rest of the page. By running the intensive workload on a Web Worker means that the website no longer needs to wait for the function to complete loading the page. This faster load time of the page means that the user can quicker read the text.

Our intensive workload simply creates an element on the web page that says 'done'. Without Web Workers this 'done' message will be visible when the page has completed loading, while with Web Workers the message will appear long after the page has completed loading.

Chapter 4

Literature Review

When searching for related work we used keywords such as: Web Workers, JavaScript, and Multi-threading on Web applications. During our research we found two papers that were related to our work and Web Workers in JavaScript.

The first paper made a performance scalability analysis with Web Workers. Their main area of research was to find the optimal number of workers for three different use cases, using single-threaded and multi-threaded multi-cores CPUs, as well as, comparing against Chrome, Firefox and Internet Explorer. Their conclusion was that current approaches do not equip the developers with enough information to estimate the optimal number of workers. Factors that affected the result were the specific use case, the users' underlying CPU architecture and even which web browser that was used. In their experiments spawning a large number of workers did not yield better performance. The authors concluded that the best method to determine the ideal number of workers is to use performance monitoring that can detect real time changes in shared resources and if workers are overloading the CPU [27].

The last paper was to help load distribution by using Web Workers for a real-time web based MORPG game application. Their main area of research was to find a solution to distribute some of the CPU workload from their web server to the clients that were currently using the web application. The authors found that assigning some caching components to the users suppresses the increase in communication workload, but also that using Web Workers the CGI latency decreased both on low-end and high-end servers with an average of 59.5% [28].

We found no papers related to responsiveness or common use cases that developers could run into. Web Workers main purpose is to improve user experience and responsiveness but despite this, no papers have confirmed this. Common use cases and responsiveness is what we want to explore and research. This gives our paper value and originality compared to related work.

4.1 What is a Web Worker?

Web Workers is a Web API that makes it possible to let web content run scripts in background threads parallel to their main thread. The background thread can complete tasks without interrupting the user interface. Web Workers use message-passing as the communication mechanism [22]. The messages are sent between the main thread and the worker using the `postMessage()` method and received with the `onmessage` event handler. The message is always copied and never shared [29].

It exists three different kinds of workers

- Dedicated Workers
  - Is only accessible by the script that called it.
- Shared Workers
  - Is accessible by multiple scripts by communicating via a port object.
- Service Workers
  - Sits between the application and the network and acts like a proxy server to enable effective offline experiences.

We will only use dedicated workers in our thesis as both shared workers and service workers are outside of our scope.

4.1.1 Thread safety

A race condition is an undesirable situation that take place when the computer perform two or more operations in parallel, but the operations must be done in the proper sequence to achieve the desired result [30].
A thread-safe library guarantees to solve the race condition problem even when it is used by multiple threads concurrently [31]. There are various strategies to avoid race conditions.
Web Workers solve this by sharing data using a message-passing strategy that serializes the messages, but as well is only allowing thread-safe components to be sent to other threads. This means that components that are not thread-safe and the Document Object Model (DOM)[8] are not allowed to be accessed on other threads except the main thread [29].

4.1.2 Example

This example is rather trivial, but will introduce you to the basic concepts that are used with workers. We will send two numbers and the background thread will multiply them and return the result.

To spawn a new worker is simple. Call the `Worker()` constructor and the path to the script that will be executed in the worker thread. This is done on the main thread (main.js):

```
let myWorker = new Worker('worker.js');
```

To start using your created worker you need to use the `postMessage()` method and send over the two values you want to multiply (main.js):

```
myWorker.postMessage([1, 2]);
```

Inside the worker script you listen on the `onmessage` event handler, and inside you complete your task and send a new message back with the `postMessage()` method that contains the result (worker.js):

```
onmessage = function(e) {
    let result = e.data[0] * e.data[1];
    postMessage(result);
}
```

Back in the main thread, to receive the message from our worker we use the `onmessage` event handler on `myWorker` (main.js):

```
myWorker.onmessage = function(e) {
  let result = e.data;
  console.log("1 * 2 = " +  result);
}
```

After running this example you have successfully calculated multiplication on a background thread using Web Workers Web API and your browser will print the result inside the console.

4.1.3 Improvements

With multi-core CPUs becoming widespread, one way to increase performance is to split the workload between multiple threads and achieve parallel computing. With Web Workers this is possible by spawning multiple workers and giving each their own chunk of the workload.

Before Web Workers, some tasks were not simply feasible to run in JavaScript. Such as long running tasks and heavy computations. The main reason these tasks were not suited for JavaScript was because one task starts to run and needs to be completed before the next task can begin. This means that the responsiveness for both the website and the user interactions would have to wait for their turn. Running these types of tasks in Web Workers instead means that the main UI thread is free to run in parallel with the worker, and the responsiveness is no longer sharing resources with the running task [32].

4.1.4 Limitations

As discussed above, to keep Web Workers thread-safe, one big limitation is that workers do not have access to the DOM, which means that a worker can not modify or read the elements inside the HTML document. This needs to be done indirectly by sending messages back to the main thread and executing the action there. Furthermore, global variables and functions inside the main thread are not obtainable for the Web Workers.

Web Workers have a high performance start-up cost and a high per-unit memory cost. The lifetime of a worker is expected to be long-lived [22]. This makes tasks that complete quickly, as well as, where a high number of workers are used unsuitable for Web Workers. For example, it would be inefficient to use Web Workers to complete simple math equations if the completion time is fast. Likewise, it would be unsuitable to spawn one worker for each pixel of an image.

Chapter 5

Results and Analysis

5.1 Experiment results

5.1.1 Prime numbers

The X axle is in milliseconds and Y axle shows the number of workers used. The baseline is based on zero workers running in Chrome as a result of having the shortest completion time with zero workers. The results from both Firefox and Chrome is then compared against the baseline to give the difference in percent.

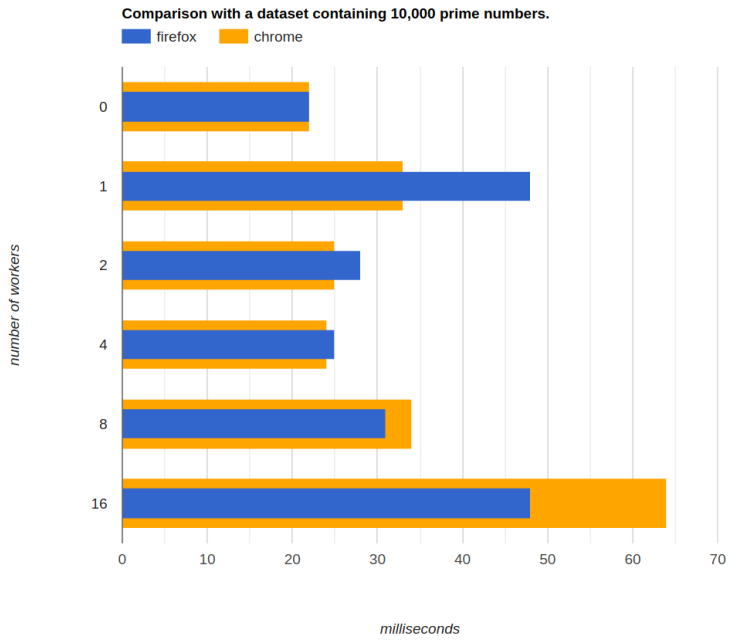

Figure 5.1: 10,000 prime numbers

Workers	Firefox		Chrome	
0	22ms	100%	22ms	100%
1	48ms	218%	33ms	150%
2	28ms	130%	25ms	111%
4	25ms	114%	24ms	109%
8	31ms	141%	34ms	155%
16	48ms	218%	64ms	291%

Table 5.2: 10,000 prime numbers

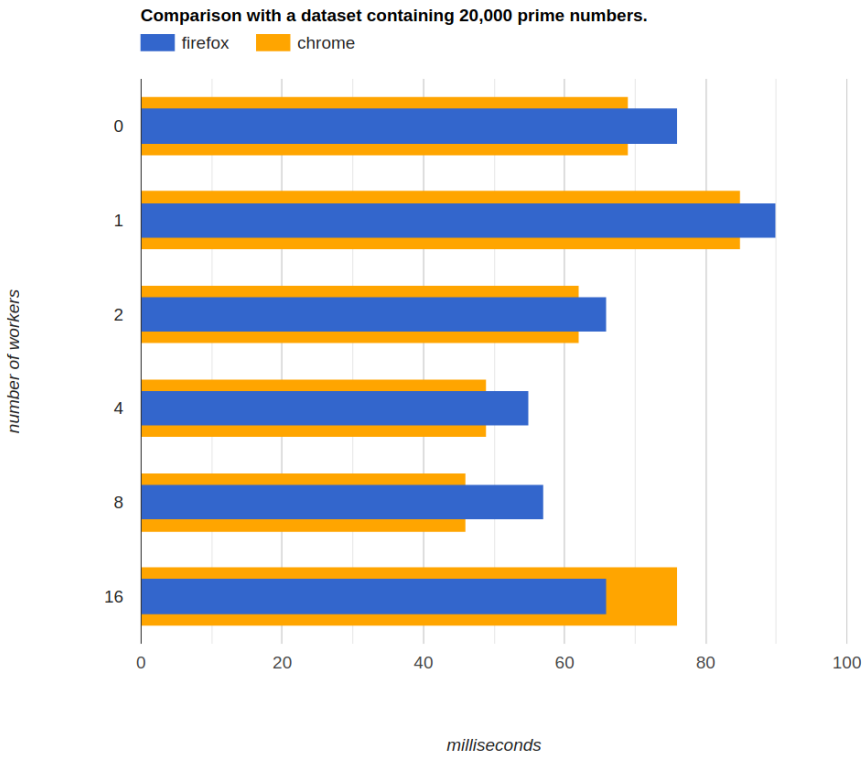

Figure 5.3: 20,000 prime numbers

Workers	Firefox		Chrome	
0	76ms	110%	69ms	100%
1	90ms	130%	85ms	123%
2	66ms	96%	62ms	90%
4	55ms	80%	49ms	71%
8	57ms	83%	46ms	67%
16	66ms	96%	76ms	110%

Table 5.4: 20,000 prime numbers

In both figure 5.1 and figure 5.3 we see a difference between the web browsers' completion times in our graphs.
The quickest completion time for the two web browsers in figure 5.1 is to use zero workers. The slowest completion time was with sixteen workers. This is the result of the high start-up cost and message passing bottleneck for individual workers. In figure 5.3 the quickest completion time is using four workers in Firefox and eight workers in chrome.
Firefox is only faster than Chrome when the calculation is done with eight and sixteen workers in figure 5.1 and sixteen workers in figure 5.3. The completion time differs at most around 16 milliseconds between the browsers in figure 5.1 and 11 milliseconds in figure 5.3.

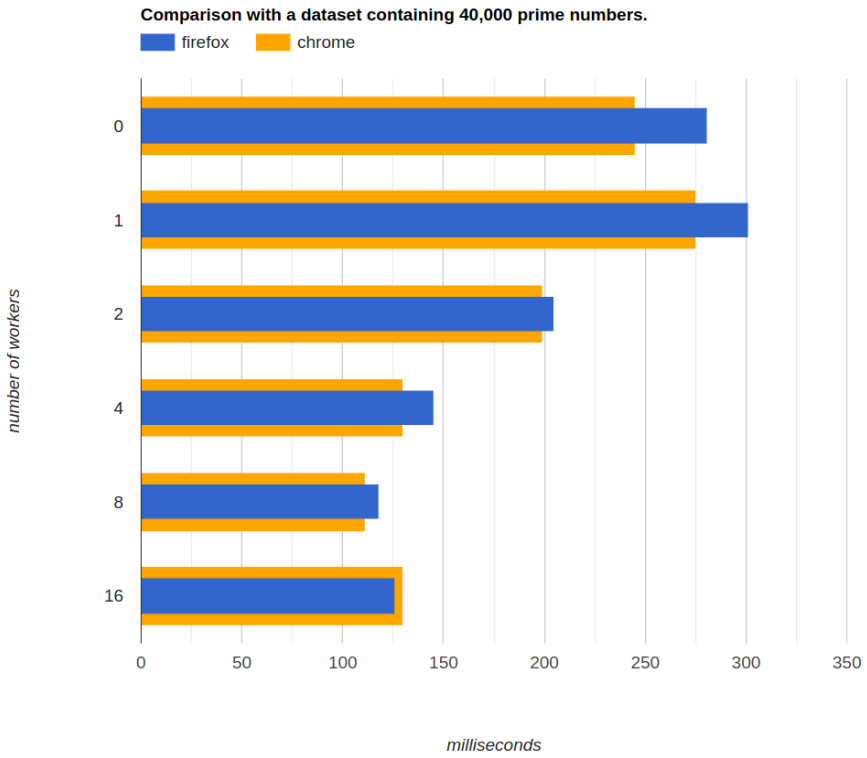

Figure 5.5: 40,000 prime numbers

Workers	Firefox		Chrome	
0	281ms	115%	245ms	100%
1	301ms	123%	275ms	112%
2	205ms	84%	199ms	81%
4	145ms	59%	130ms	53%
8	118ms	48%	111ms	45%
16	126ms	51%	130ms	53%

Table 5.6: 40,000 prime numbers

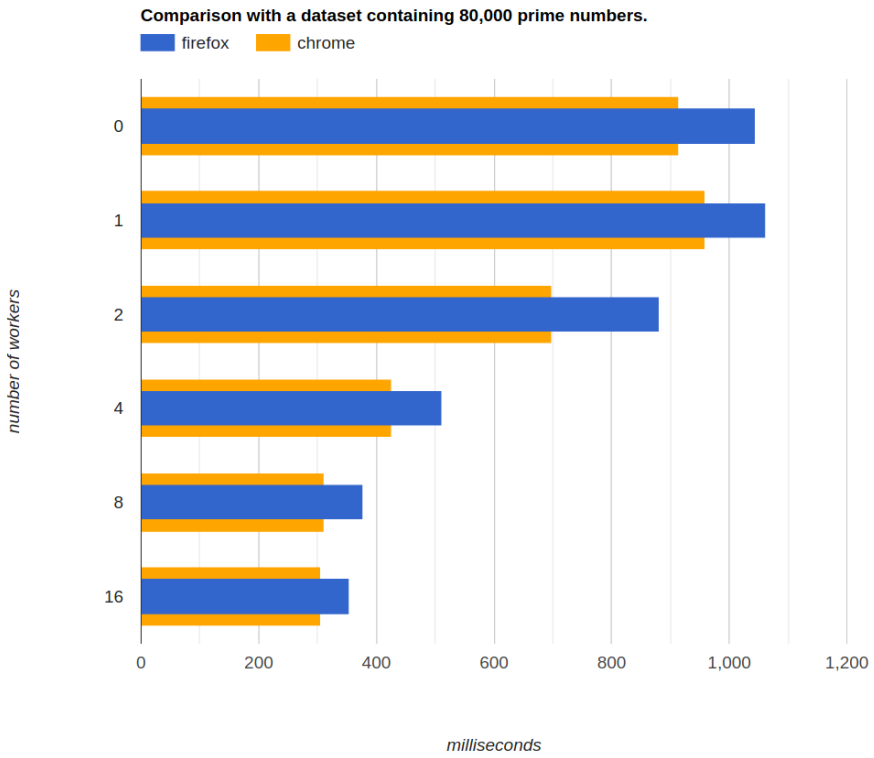

Figure 5.7: 80,000 prime numbers

Workers	Firefox		Chrome	
0	1045ms	114%	915ms	100%
1	1062ms	116%	958ms	105%
2	881ms	96%	698ms	76%
4	511ms	56%	426ms	47%
8	378ms	41%	311ms	34%
16	354ms	39%	305ms	33%

Table 5.8: 80,000 prime numbers

Figure 5.5 and figure 5.7 both tell a similar story with a clear advantage of using multiple workers. In Chrome, eight workers were the fastest in figure 5.5, and sixteen workers in figure 5.7. In Firefox, eight workers were faster in figure 5.5, and sixteen workers in figure 5.7. The slowest execution time for both browsers was using one worker, which makes sense because its completion time should be the same as zero worker, but with added start-up cost and the overhead with message passing between the main thread and the worker thread.
Again, Chrome was faster than Firefox in the main thread with zero workers. Chrome was quicker in all scenarios except one in figure 5.5. That scenario was using sixteen workers. In figure 5.7 all the scenarios were in favor of Chrome. The completion time differs at most around 36 milliseconds between the browsers in figure 5.5 and 183 milliseconds in figure 5.7.

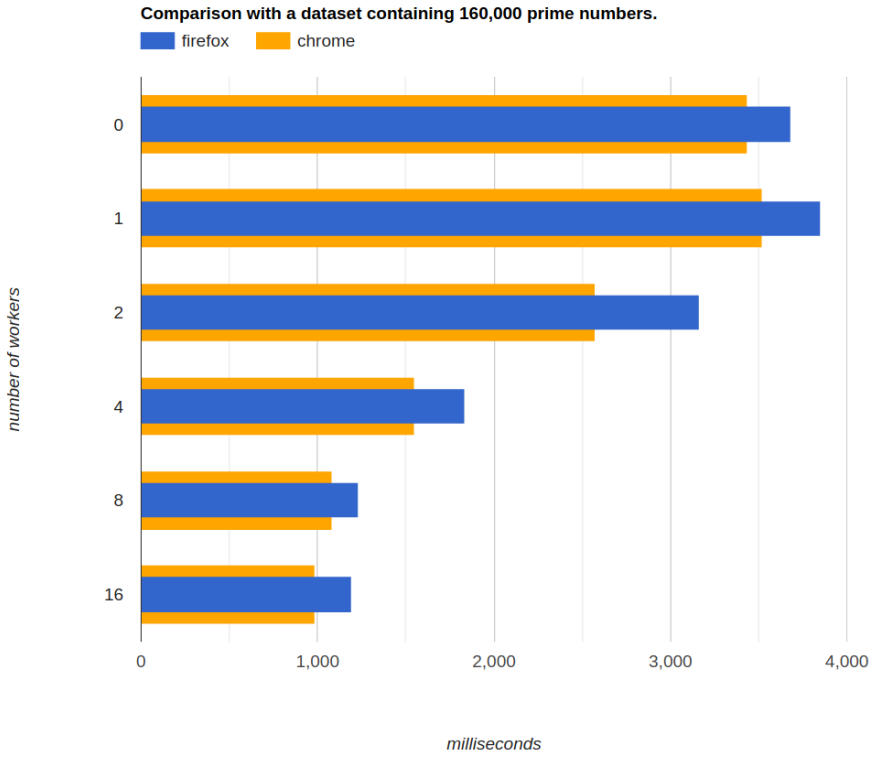

Figure 5.9: 160,000 prime numbers

Workers	Firefox		Chrome	
0	3,7s	109%	3,4s	100%
1	4,9s	144%	3,5s	103%
2	3,2s	94%	2,6s	76%
4	1,8s	53%	1,6s	47%
8	1,2s	35%	1,1s	32%
16	1,2s	35%	1,0s	29%

Table 5.10: 160,000 prime numbers

The fastest execution time was achieved with sixteen workers in both Firefox and Chrome. Only a 3% improvement (100 milliseconds) was found in Chrome using sixteen workers versus eight. With Firefox an insignificant improvement below 1% (42 milliseconds) was found using sixteen workers versus eight. Chrome is faster than Firefox in all scenarios. Here we can see the benefits of using multiple threads and with sixteen workers the completion time has decreased with 74% for Firefox and 71% for Chrome.

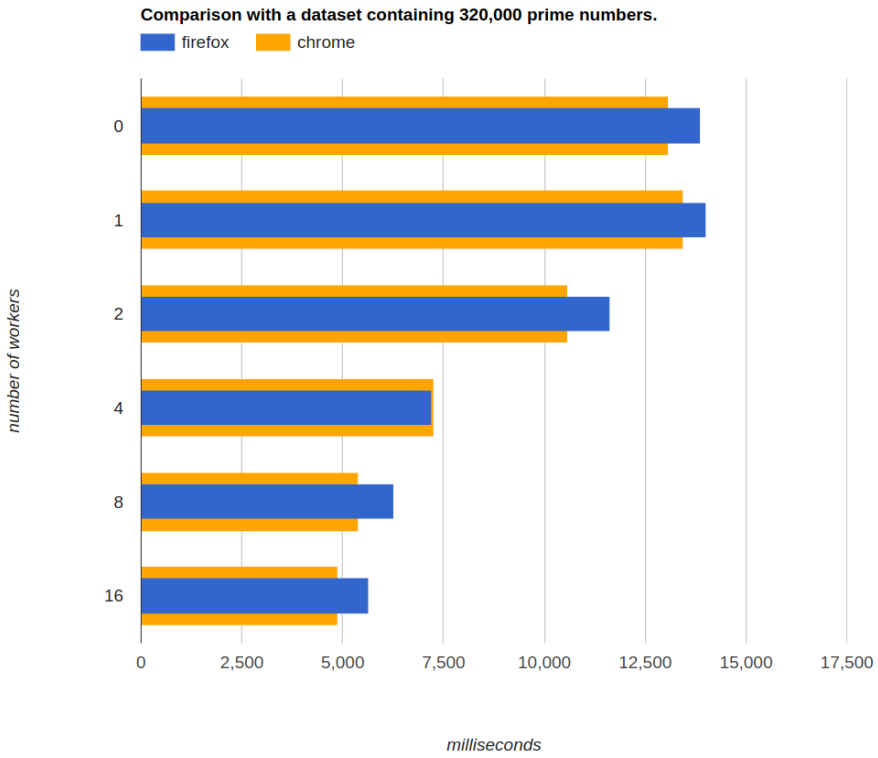

Figure 5.11: 320,000 prime numbers

Workers	Firefox		Chrome	
0	13,9s	106%	13,1s	100%
1	14,0s	107%	13,4s	99%
2	11,6s	89%	10,6s	81%
4	7,2s	55%	7,3s	56%
8	6,2s	47%	5,4s	41%
16	5,6s	43%	4,9s	37%

Table 5.12: 320,000 prime numbers

The fastest execution time was sixteen workers for both web browsers. The difference between the web browsers' completion times are small but Chrome is faster in all the scenarios, except with four workers where Firefox is 1% (100 milliseconds) faster. A 4% improvement (500 milliseconds - 600 milliseconds) is found when using sixteen workers instead of eight in both browsers. A 15% improvement (1.9 seconds) is found in Chrome when switching from four workers to eight workers. In Firefox, 8% quicker completion time (1 second) is measured with eight workers against four workers.

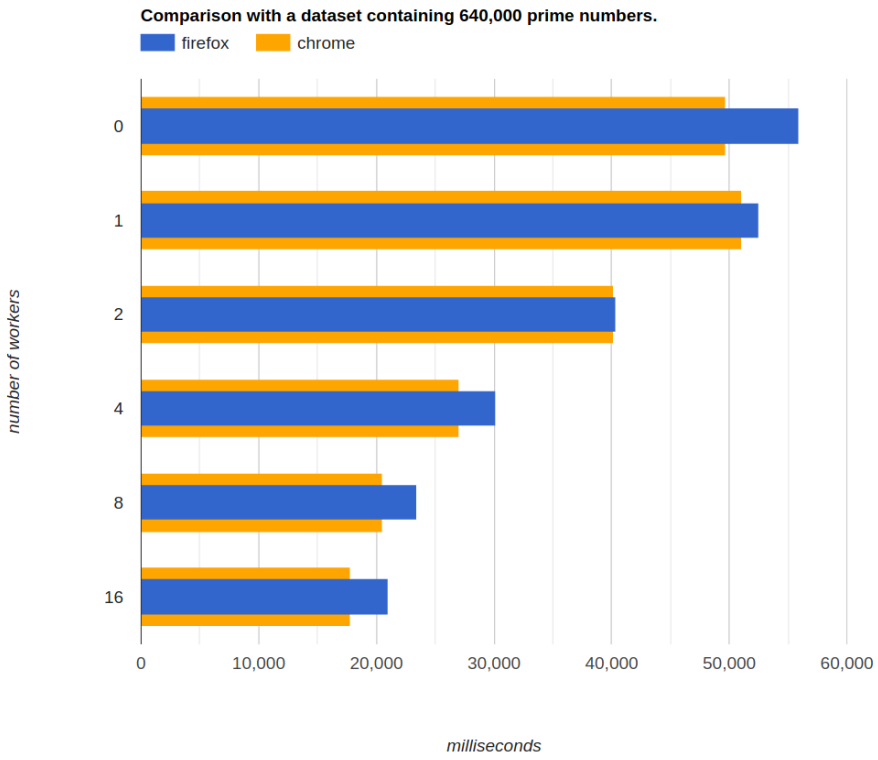

Figure 5.13: 640,000 prime numbers

Workers	Firefox		Chrome	
0	55,9s	112%	49,7s	100%
1	52,5s	106%	51,1s	103%
2	40,3s	81%	40,2s	81%
4	30,1s	61%	26,9s	54%
8	23,5s	47%	21,5s	41%
16	21s	42%	17,8s	36%

Table 5.14: 640,000 prime numbers

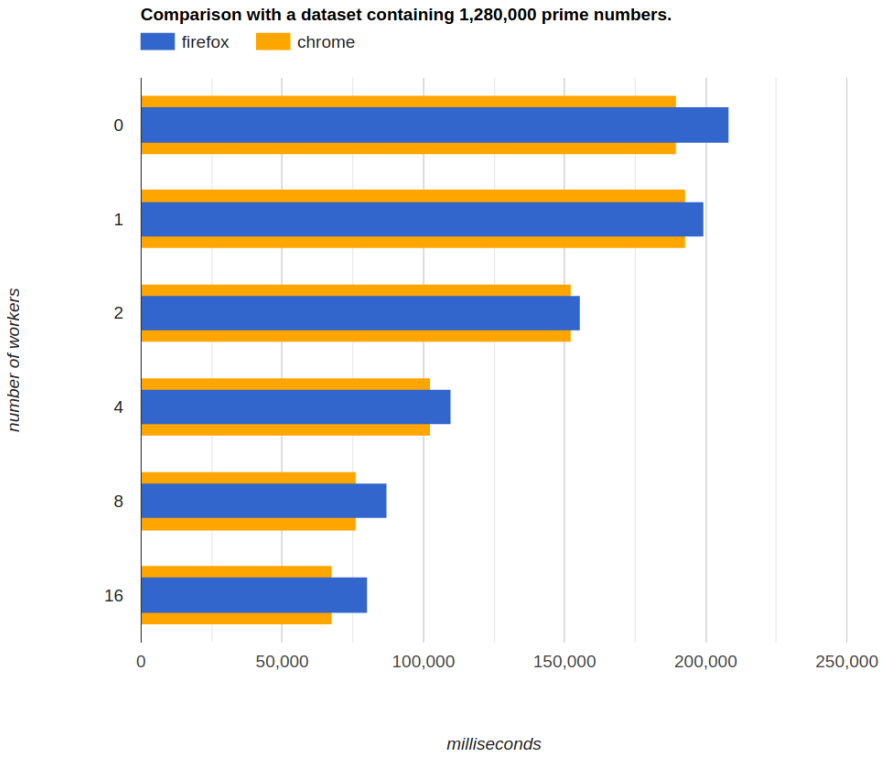

Figure 5.15: 1,280,000 prime numbers

Workers	Firefox			Chrome		
0	208s	3,5min	109%	190s	3,2min	100%
1	199s	3,3min	103%	193s	3,2min	100%
2	155s	2,6min	81%	152s	2,5min	78%
4	110s	2,0min	63%	102s	1,7min	53%
8	87s	1,8min	56%	76s	1,3min	41%
16	80s	1,5min	47%	68s	1,1min	34%

Table 5.16: 1,280,000 prime numbers

The results from figure 5.13 and figure 5.15 are similar when it comes to the difference in number of workers and completion time in percent. Sixteen workers were the fastest for both browsers. In figure 5.15 Web Workers were able to decrease the completion time from around three and half minutes without workers to around one and a half minute with sixteen workers (62%) in Firefox. In Chrome, the completion time decreased from around three minutes without workers to one minute with sixteen workers (66%). The jump from eight to sixteen workers gave a 5% (around 3 seconds) improvement in figure 5.13 and 7% - 9% (7 seconds) improvement in figure 5.15. Chrome was faster than Firefox in all scenarios in both figure 5.13 and 5.15.

Summary

The outcome from the prime number experiment confirmed our expectations that more workers would result in faster completion time, with larger amounts of data and slower with smaller amounts. In our smallest data size (figure 5.1) where the completion time was below 50 milliseconds we saw that Web Workers' start-up cost and message passing overhead was too great. This made not using Web Workers at all have the fastest completion time and sixteen workers the slowest.
When the completion time increased in all figures, except figure 5.1, increasing the number of workers from one decreased the completion time with a clear distinction. At best the completion time decreased with 74% in Firefox and 71% in Chrome.

Using the largest data size with completion time in minutes (figure 5.13, figure 5.15), only a small increase was found when using sixteen workers (3 - 5 seconds) compared to eight workers. This means that the threshold of effectiveness is reached, the difference starts to diminish and the results stay almost the same, even though the amount of data is increasing.

The outcome from both browsers did differ above our expectations. Chrome was significantly faster than Firefox in all use cases with a few exceptions. The difference is not enormous but is clearly present.

A summary from the final improvements results against the baseline's completion time, and a proposal of how many workers are recommended, based on our experiments. The results are shown here:

Workers	Completion time	Baseline	
		Firefox	Chrome
0	t $<$ 50 milliseconds	0%	0%
4	50 milliseconds $<$ t $<$ 100 milliseconds	20%	29%
8	100 milliseconds $<$ t $<$ 15 seconds	57%	62%
16	t $>$ 15 seconds	56%	65%

Table 5.17: Summary from prime numbers experiment

5.1.2 Spell checking

Without Web Workers

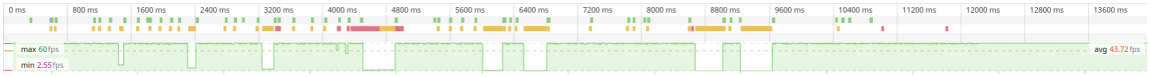

Figure 5.18: Firefox without Web Workers.

Figure 5.19: Chrome without Web Workers.

With Web Workers

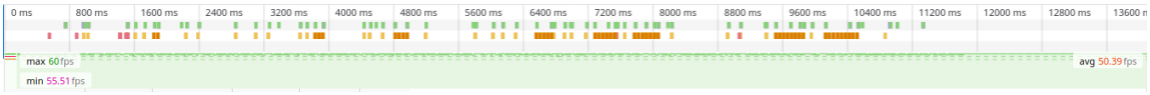

Figure 5.20: Firefox with Web Workers.

Figure 5.21: Chrome with Web Workers.

The spell checking experiment lets the user type in different words and when the space button is hit, it will evaluate the characters and compare it with the English dictionary to determine if it is a correct word or not. If the same word is typed more than once there will be a counter that shows how many times that word has been used. In Firefox the responsiveness losses are marked as holes, while in Chrome they are marked as horizontal red bars. From the results we can see that both Firefox and Chrome without Web Workers experienced drops in responsiveness multiple times during the process. While the results with Web Workers were significantly better. Both Firefox and Chrome were flawless. The experiment was constructed in a way that would make it very performance consuming. Normally, this could have been solved in other ways. However, we decided to use this approach to display the advantage with Web Workers.

Summary

The spell checker experiment performed above our expectations. The improvements were comprehensive when using Web Workers for both browsers. Without Web Workers the performance experienced drops when handling many words, while with Web Workers the performance was flawless. We did expect the results to be better with Web Workers, but not to that extent.

5.1.3 Load time

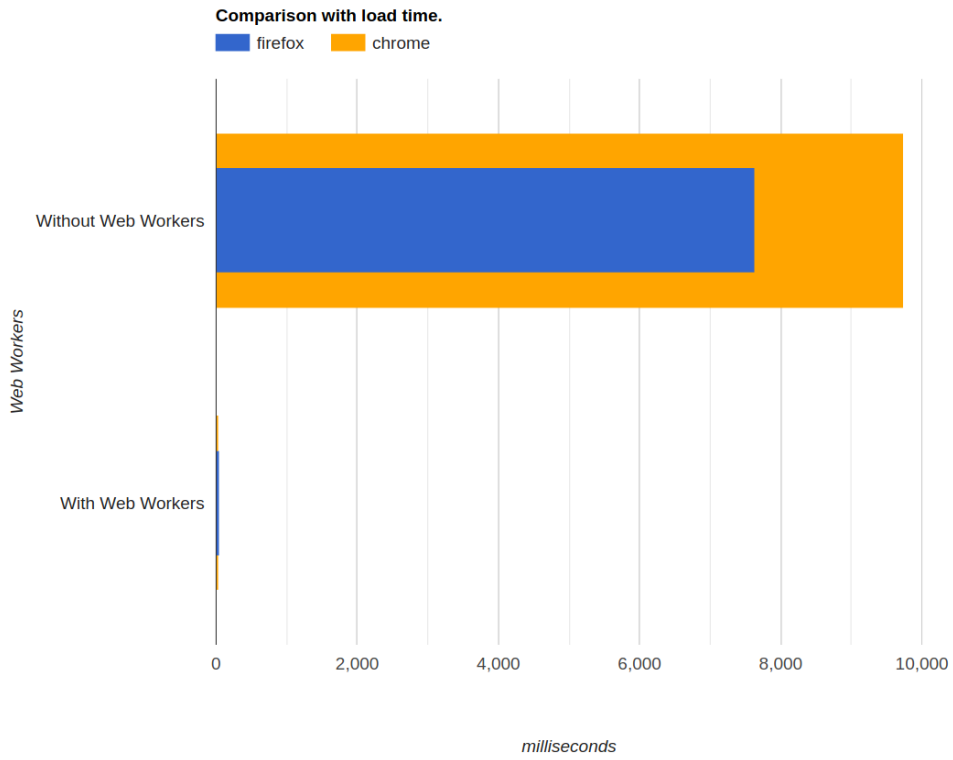

Figure 5.22: Load time.

For this experiment the load time was tested. For this test a big text file was loaded while something else was running in the background. The results between using Web Workers or not was vast. The reason why there was such a great improvement when using Web Workers is because the workers can simultaneously load the content and everything that was running in the background. Without Web Workers, the content can not be loaded until all processes have been completed [22]. Firefox was slightly faster than Chrome in this experiment.

Summary

The Load time experiment results were above our expectations. There was a vast improvement when using Web Workers for both browsers. The expected outcome was that with Web Workers the load time should be better, but not to that extent. The difference between the browsers was minimum, which confirmed our expectations.

5.2 Survey

The survey contains questions regarding developers' background, performance in web applications, multi-threaded applications, and knowledge about Web Workers. The reason why the survey includes questions about performance and multi-threaded applications is because they creates the foundation for Web Workers. If people do not experience issues with performance as it stands today, there would be no interest in introducing more complexity into their applications, in this case implementing Web Workers.

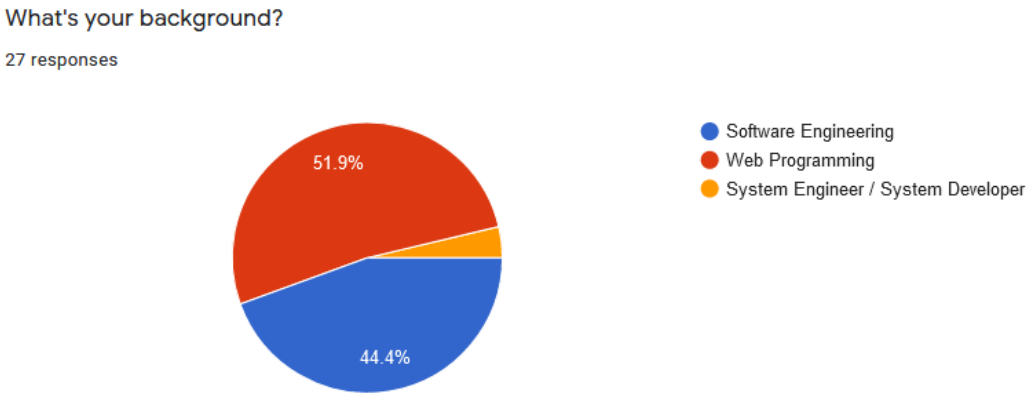

Figure 5.23: Background.

With this question we wanted to find out what background the participants have. The answers could then be evaluated to determine if background had any significance for the final results.

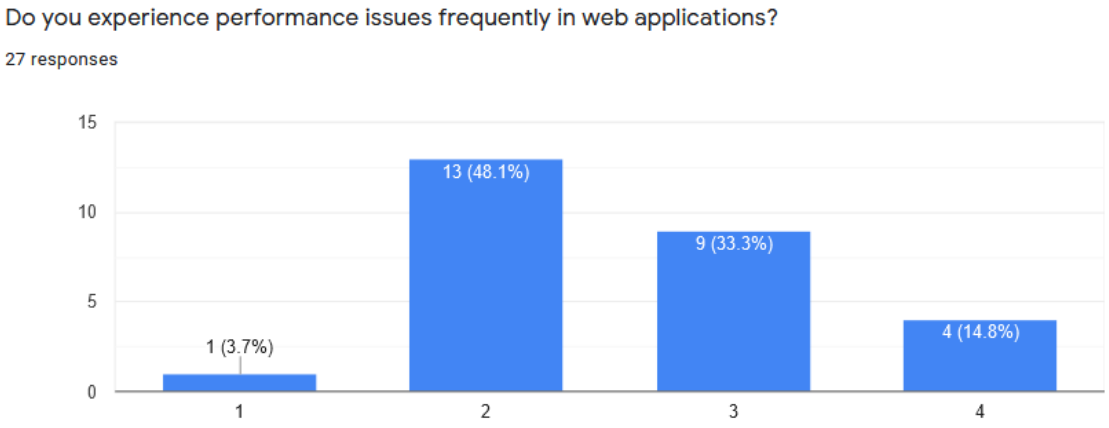

Figure 5.24: Performance issues in web applications.

With this question we wanted to see if performance issues is something people experience often in web applications. One means that experience issues do not occur frequently, two means it happens sometimes, three means more often, and four means it does frequently.

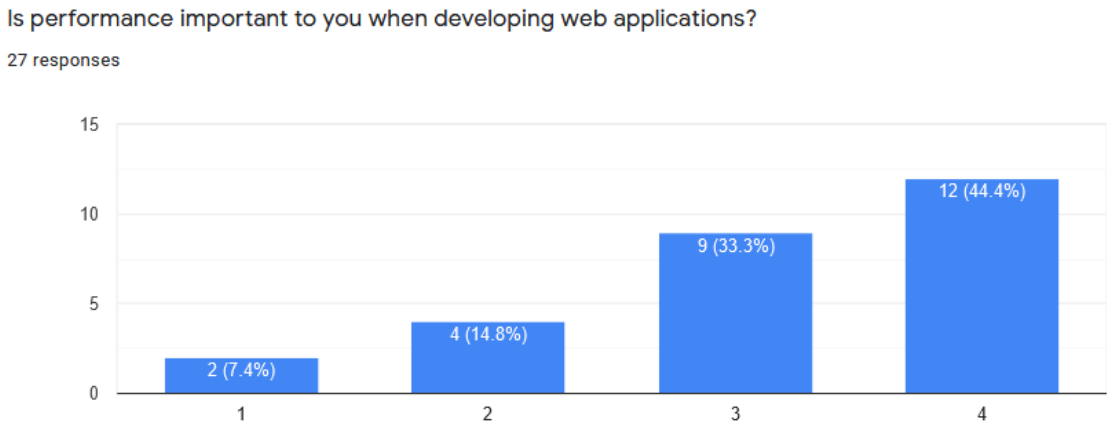

Figure 5.25: Importance of performance in web applications.

This question gives us an indication of how important performance is when developing web applications. One means that it is not very important, two means quite important, three means important, and four means it is very important.

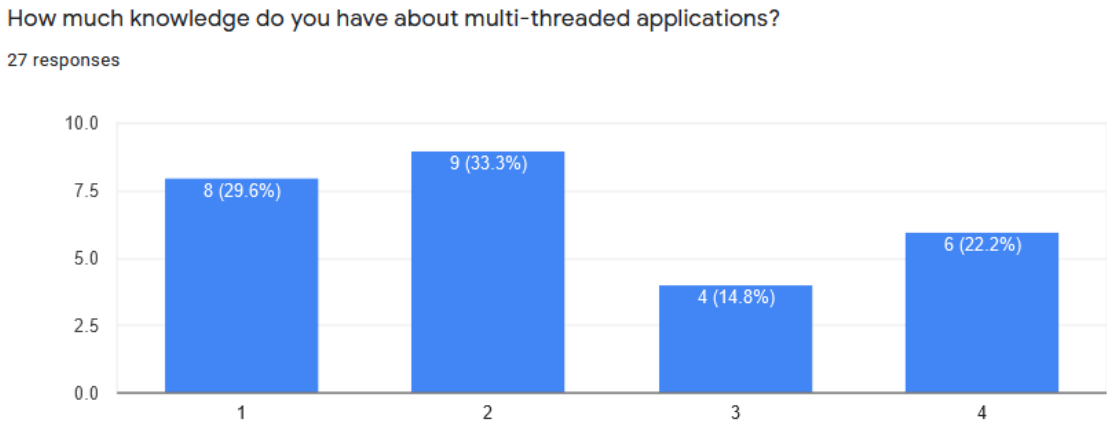

Figure 5.26: Knowledge about multi-threaded applications.

With this question we wanted to find out how much knowledge people have about multi-threaded applications. One means that not much knowledge is acquired, two means a little more, three means some knowledge, and four means much knowledge is acquired.

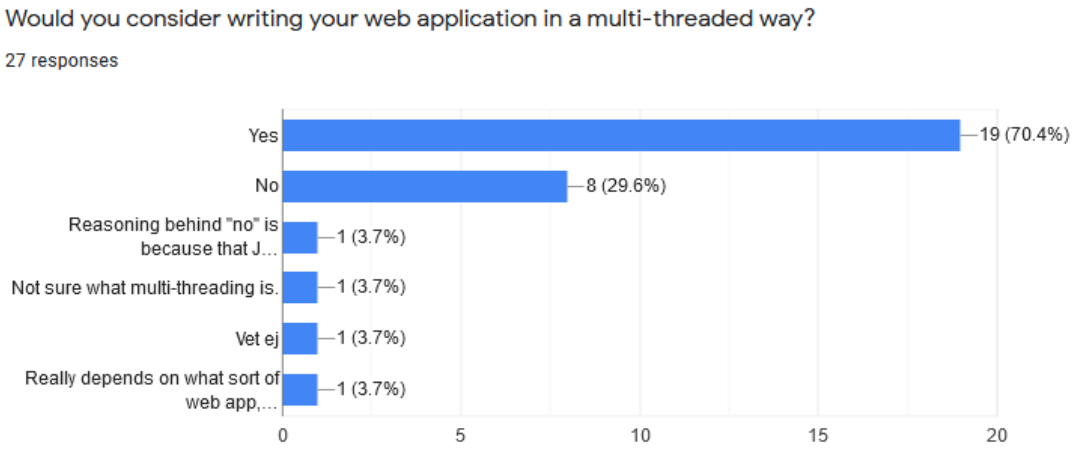

Figure 5.27: Web application in a multi threaded way.

With this question we could see if anyone would consider writing their web applications in a multi-threaded way.
Other responses were:

- Reasoning behind "no" is because that JS is built to be runned in a single thread "the event handler" and is limited to it since there is no good way of sharing the "data" between the different workers in a good/efficient way.
- Not sure what multi-threading is.
- Don't know (Vet ej).
- Really depends on what sort of web app, if I have a chance to use a language that handles multi-threading well then yes. But often you're stuck with dumb things like node or whatever.

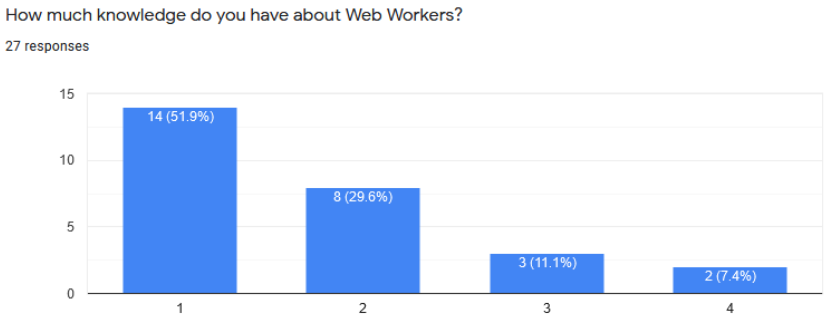

Figure 5.28: Knowledge about Web Workers.

This question was the foundation of the survey. To see if anyone had any knowledge about Web Workers. One means that not much knowledge is acquired, two means a little more, three means some knowledge, and four means much knowledge is acquired.

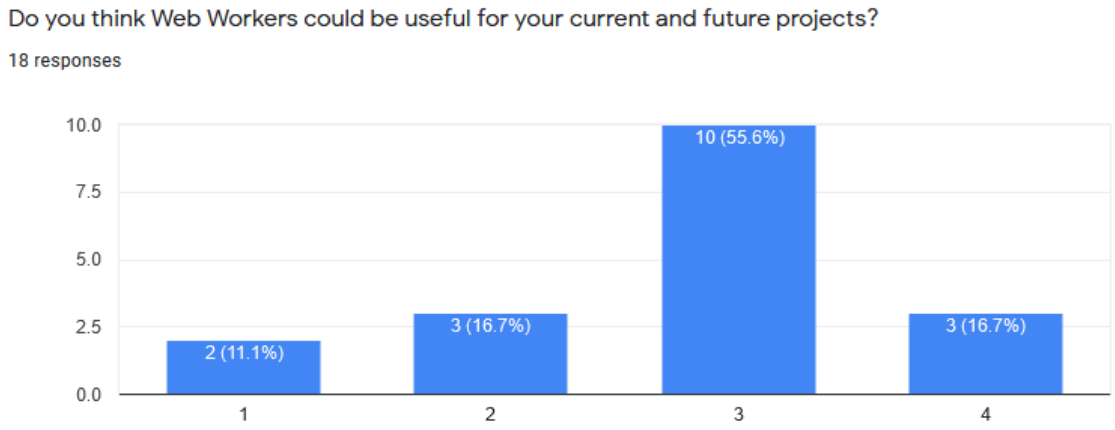

Figure 5.29: Web Workers in future projects.

With this question we wanted people's opinions on if they would consider using Web Workers in their projects. One means that Web Workers could not be that useful, two means it might, three means it would, and four means it would be very useful.

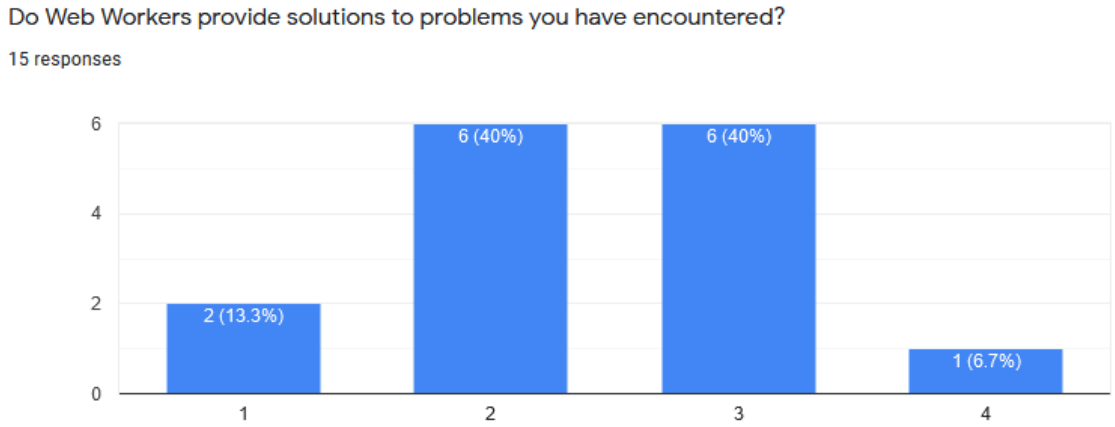

Figure 5.30: Solutions to real problems.

This question gives us insight if Web Workers could solve problems that has been encountered before. One means that Web Workers would not provide solutions to problems encountered, two means it might, three means it could, and four means it would provide solutions.

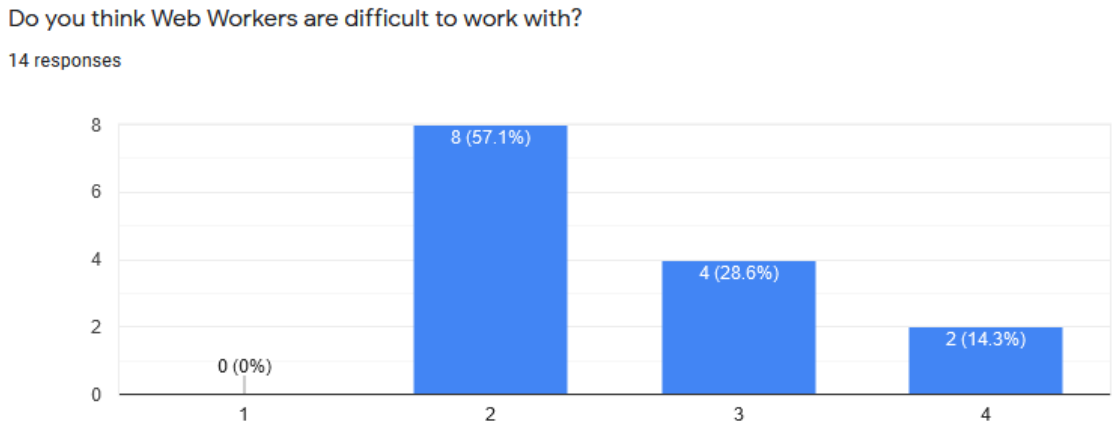

Figure 5.31: Web Workers are difficult to work with.

This question gives us an indication on if Web Workers are considered difficult to work with. One means that Web Workers are considered not difficult to work with, two means slightly difficult, three means difficult, and four means very difficult to work with.

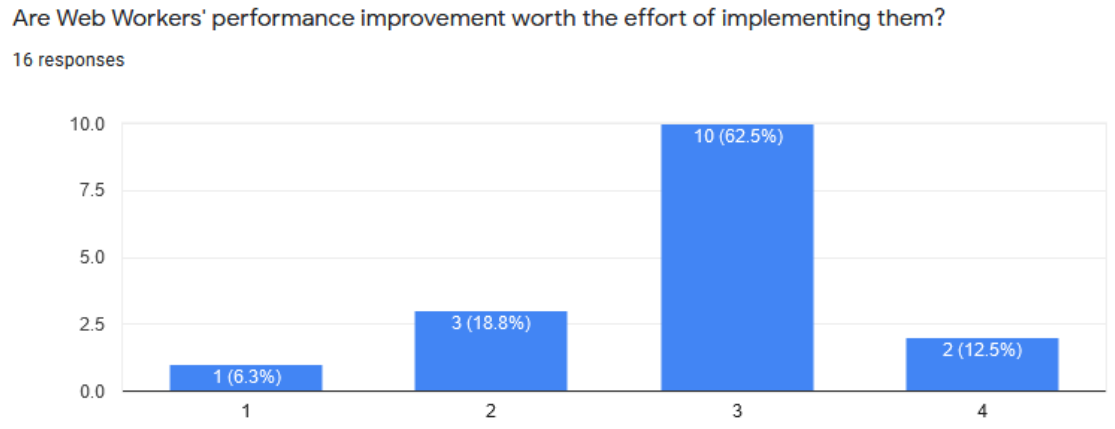

Figure 5.32: Web Workers are worth the effort to implement.

The final question gives us insight if the performance improvement would be worth the effort it takes to implement them. One means that it would not be worth the effort, two means it might be, three means it could be, and four means it would be worth the effort.

Summary

We sent out a survey to collect data on developers' knowledge and opinions about Web Workers. The total number of participants was 27. Out of the 27 responses, the majority had a Web Programming background. Not so many experience performance issues frequently in web applications, but most of them think it is important to have in mind when developing. Most of the people did not have much knowledge about multi-threaded applications, but would still consider writing an application in a multi-threaded way. The knowledge about Web Workers was low as expected, however, many believe that Web Workers may solve problems they have encountered and would find them useful in their own projects. The majority believes that Web Workers are difficult to implement, but would still consider it worth the effort.

From the responses we can conclude that there is an interest in performance when developing applications, and writing it in a multi-threaded way could be the solution. Our work proves that performance can be significantly improved with a multi-threaded implementation. By using Web Workers in web applications, it provides faster load time, better responsiveness, and increases raw computational power. This conclusion may convince people that multi-threading is advantageous for applications where performance is a high priority. In this case the focus was on Web Workers in JavaScript and the improvements it provides for web applications. Since JavaScript is a commonly used programming language for web developers, our conclusion could give web developers more insight in the benefits of multi-threading and hopefully bring a new way of thinking.

5.3 Analysis

Based on the results from the experiments we have clearly shown how Web Workers can significantly improve responsiveness, raw computational power and load time. In use cases where responsiveness is the central point, Web Workers made a massive difference. From an unresponsive user experience, to perfectly smooth typing responsiveness by simply moving the heavy process to a background thread is astonishing. Personally, we will always implement Web Workers when working on future projects where responsiveness is important.
Web Workers still have some limitations about responsiveness. The biggest one is when responsiveness is depending on waiting for the return value from the previous action before the next action can be executed. Using Web Workers in these kinds of use cases will worsen the response time. That said, using Web Workers for these kinds of tasks will still give an improvement in responsiveness for user inputs, such as mouse and keyboard.

In use cases where parallel computing is possible, using Web Workers can truly decrease the completion time in half or more. These use cases need to be available to divide the problem into smaller tasks that can be solved simultaneously.
The results obtained from our experiments confirm that Web Workers have a high start-up cost, as mentioned in the literature review, and should not be used when tasks have a quick completion time. On the other hand, the start-up cost is lower than anticipated and based on our testing, a task with completion time around 100 milliseconds could be a viable option, in order to be optimized with Web Workers.

For tasks with completion time between 100 milliseconds and one second, we would generally recommend using four or eight workers. Eight workers are recommended when it is over one second, and sixteen workers when it is over 15 seconds. This is based on our prime number experiment. Every use case is different and some testing is needed to find the best performance gain.

The load time experiment shows one specific use case where Web Workers do make a compelling difference. However, this is only a very specific scenario and during our initial testing with different use cases, Web Workers often did not make a significant difference. To distinguish a big difference between using Web Workers and not, the specific use case we found was running an intensive workload in the background.

Chapter 6

Conclusion

The performance results from the two web browsers shifted depending on which experiment that was executed. The prime number experiment was in favor of Chrome with and without Web Workers, with a few exceptions. The spell checking experiment gave the same results with and without Web Workers in the two browsers. The load time experiment was in favor of Firefox without Web Workers, but even with Chrome when Web Workers were used. Those results will answer how much performance differs between Firefox and Chrome when using Web Workers.

To find the threshold of effectiveness for Web Workers, the prime number experiment illustrates how Web Workers can improve raw computational power and where the threshold lies in the number of workers. Our results confirm that using Web Workers is beneficial when processing large amounts of data in use cases where the workload is spliced into a parallel workflow. From the results, it would be beneficial to use eight workers if your task takes between 100 milliseconds and 1 second on the main thread. This gave us at best an improvement of 67% quicker completion time. For tasks that take longer than 15 second, it would be favorable to use sixteen workers instead. Using one worker will consistently be slower than the main thread, because of the high start-up cost and the overhead of message passing. Using more than eight workers only improves the completion time when 80,0000 or larger data set size is being used.
The spell checking experiment further answers the threshold of effectiveness for Web Workers and displays how responsiveness is significantly improved when Web Workers are being used. It is beneficial to use Web Workers for the majority of processes when applications are used extensively by the user at fast intervals, and when responsiveness is important for the user experience. The small performance impact is worth the gain in responsiveness and no threshold was found where Web Workers' responsiveness started to diminish.
The load time experiment displays how useful Web Workers can be when loading lots of resources on a web page. Those results give a strong indication of when Web Workers could be considered worth implementing. To find the threshold of effectiveness for Web Workers in this experiment was hard, because the difference between using few workers and multiple workers made no distinction.

Developers' opinions were obtained by the responses from the survey. Those answers gave us more insight in how much knowledge software developers have about multi-threaded applications and specifically Web Workers. This confirmed our expectations that the majority of participants had none or little knowledge about Web Workers, but would be interested to work with them in current and future projects. The reason for that may be the potential performance gain that could be acquired. The responses gave us answers to what developers' opinions about Web Workers are.

Chapter 7

Validity Threats

The experiments are running inside an isolated docker container that hosts the web server to minimize external factors, however, client-side JavaScript still needs to run on the user operating system. Which means the outcome may be affected if the user is running multiple applications simultaneously as doing the experiments, e.g. having many tabs open in the web browser.

The tests were executed in Chrome and Firefox. Thus, the results can not be transferred to other web browsers, but will give an indication of the expected outcome in them.

Another factor could be that the results may differ depending on where the experiments are running, because of different hardware. In addition to that, there may be a case when the hardware required is not fulfilled. We did only run the experiments on one computer with a specific hardware configuration. To draw a more accurate conclusion we could have run the experiments on different hardware and compared the results against each other.

For collecting information about developers' opinions about Web Workers we relied entirely on a high response rate from the survey. If the response rate for the survey is poor, it becomes hard to come to a fair conclusion on developers' opinions about Web Workers. We could have posted the survey on more web sites and sent it out to more people, in order to prevent that from happening. When sharing it on multiple web sites and with more people the decreasing risk of poor response rate is one reason. Another reason is that it reaches out to developers all over the world, not only in Sweden. That may result in different thoughts and opinions.

Chapter 8

Future Work

Follow-up for this study could have been to do a comparison between the different kinds of workers. Do a profound analysis on the different attributes they acquire and how their performance result differs. Find use cases when one kind of worker is more suitable than another. There could have been questions in the survey what developers think about the different workers and when they are applicable.

Another follow-up for this thesis could have been to run experiments on different hardware. The first execution could run on a slower CPU and the next on a faster one. The data could then be collected and analyzed to determine how much impact hardware has. The experiments we did were executed using docker, which means they run inside a docker container, isolated from external configurations. The tests could have been compared when running inside a container and when running outside to see if the results are changing.

Since this thesis is focused on client-side JavaScript, a follow-up could be to shift focus to server-side JavaScript instead. Run experiments and collect test results to see how performance in server-side JavaScript differs in comparison to client-side JavaScript.

References

[1] Andrew S. Tanenbaum. *Modern Operating Systems 4th Edition*, chapter 2.2.2 The Classical Thread Model, pages 103–104. Pearson, 2015.

[2] Parallax Inc. *Single Core vs Multicore*, 2020. `https://learn.parallax.com/tutorials/language/propeller-c/propeller-brains-your-inventions/single-core-vs-multicore` [2020-06-02].

[3] MDN. *What is a web browser?*, 2020. `https://www.mozilla.org/en-US/firefox/browsers/what-is-a-browser/` [2020-06-02].

[4] Ali Alakeel. A guide to dynamic load balancing in distributed computer systems. *International Journal of Computer Science and Network Security (IJCSNS)*, 10, 2009.

[5] Tech Differences. *Difference Between Server-side Scripting and Client-side Scripting*, 2018. `https://techdifferences.com/difference-between-server-side-scripting-and-client` `html` [2020-06-02].

[6] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., USA, 1989.

[7] MDN. *Introduction to web APIs*, 2020. `https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction` [2020-06-02].

[8] MDN. *Introduction to the DOM*, 2020. `https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction` [2020-04-17].

[9] Docker inc. *What is a Container?*, 2020. `https://www.docker.com/resources/what-container` [2020-06-02].

[10] Docker inc. *The Docker platform*, 2020. `https://docs.docker.com/get-started/overview/` [2020-06-02].

[11] International Software Testing Qualifications Board. *effectiveness*, 2018. `https://glossary.istqb.org/en/search/effectiveness` [2020-06-01].

[12] Netscape Communications Corporation. Netscape and sun announce javascript, the open, cross-platform object scripting language for enterprise networks and the internet. *Netscape Communications Corporation*, 1995. https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html [2020-02-25].

[13] Souvik Banerjee. Top 10 programming languages for web development. *RS Web Solutions*, 2015. https://www.rswebsols.com/tutorials/programming/top-10-programming-languages-web-development [2020-02-25].

[14] Eliot Eshelman. Xeon e5-2600v3 number of cpu cores. *Microway*, 2014. https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/xeon_e5-2600v3_cpu_number_of_cores/ [2020-02-21].

[15] Jeff Preshing. A look back at single-threaded cpu performance. *preshing*, 2012. https://preshing.com/20120208/a-look-back-at-single-threaded-cpu-performance/ [2020-05-7].

[16] Philip Roberts. *What the heck is the event loop anyway?*, 2014. https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=896s/ [2020-04-02].

[17] Lawrence Livermore National Laboratory Blaise Barney. *POSIX Threads Programming*, 2020. https://computing.llnl.gov/tutorials/pthreads/ [2020-05-07].

[18] Chaitanya Singh. *Threads*, 2013. https://beginnersbook.com/2013/03/java-threads/ [2020-05-07].

[19] Golang organization. *Concurrency*, 2020. https://golang.org/doc/effective_go.html#concurrency [2020-05-07].

[20] Python Software Foundation. *threading — Thread-based parallelism*, 2020. https://docs.python.org/3/library/threading.html [2020-05-07].

[21] A. Taivalsaari, T. Mikkonen, D. Ingalls, and K. Palacz. Web browser as an application platform. In *2008 34th Euromicro Conference Software Engineering and Advanced Applications*, pages 293–302. IEEE, 2008.

[22] The World Wide Web Consortium. *Web Workers*, 2015. https://www.w3.org/TR/workers/ [2020-02-21].

[23] StatCounter. *Desktop Browser Market Share Worldwide*, 2020. https://gs.statcounter.com/browser-market-share/desktop/worldwide#monthly-202005-202005-bar [2020-06-01].

[24] Jakob Nielsen. *Usability Engineering*, chapter 5. Morgan Kaufmann, San Francisco, 1993. ISBN: 0-12-518406-9.

[25] MDN. *Performance*, 2019. https://developer.mozilla.org/en-US/docs/Tools/Performance [2020-02-21].

[26] Kayce Basques. *Get Started With Analyzing Runtime Performance*, 2019. https://developers.google.com/web/tools/chrome-devtools/evaluate-performance [2020-04-16].

[27] J. Verdú and A. Pajuelo. Performance scalability analysis of javascript applications with web workers. *IEEE Computer Architecture Letters*, 15(2):105–108, 2016. https://ieeexplore.ieee.org/abstract/document/7307120 [2020-05-07].

[28] Masaki Kohana Shusuke Okamoto. Load distribution by using web workers for a real-time web application. *International Journal of Web Information Systems*, 2011. https://www.emerald.com/insight/content/doi/10.1108/17440081111187565/full/html [2020-05-07].

[29] MDN. *Using Web Workers*, 2020. https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers [2020-04-16].

[30] Andrew S. Tanenbaum. *Modern Operating Systems 4th Edition*, chapter 2.3.1 Race Conditions, pages 119–121. Pearson, 2015.

[31] Oracle and/or its affiliates. *Thread Safety*, 2019. https://docs.oracle.com/cd/E37838_01/html/E61057/compat-14994.html#scrolltoc [2020-04-17].

[32] MDN. *Intensive JavaScript*, 2019. https://developer.mozilla.org/en-US/docs/Tools/Performance/Scenarios/Intensive_JavaScript [2020-04-17].

Chapter 9

Supplemental Information

Software and hardware configuration that was used when running the experiments.

- Software
  - Mozilla Firefox version 76.0.1
  - Google Chrome version 78.0.3904.97
  - Docker version 19.03.9, build 9d988398e7
  - docker-compose version 1.22.0, build f46880fe
  - YML version 3
  - Dockerfile: Apache2 → httpd:2.4
- Hardware
  - Type of Processor: i5-8265U
  - Memory Size: 8 GB
  - Storage Capacity: 256 GB
  - Graphics Processor: Intel UHD Graphics 620

Other:
Link to the source code for our experiments and docker files:

`https://github.com/JohanDjarvKarltorp/Web-Workers`

To replicate and run our experiments:

- `docker run -p 8080:80 johandjarvkarltorp/web-workers`
- Visit `http://localhost:8080/`

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden