EMPIRICAL ANALYSIS ON JAVASCRIPT ENGINE PERFORMANCE

Mohottige Don Thilina Dulanjana

179317N

M.Sc. in Computer Science

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

August 2022

EMPIRICAL ANALYSIS ON JAVASCRIPT ENGINE PERFORMANCE

Mohottige Don Thilina Dulanjana

179317N

Thesis submitted in partial fulfillment of the requirements for the degree

M.Sc. in Computer Science

Department of Computer Science and Engineering

Faculty of Engineering

University of Moratuwa

Sri Lanka

August 2022

DECLARATION

I declare that this is my own work and this thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any other University or Institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant to University of Moratuwa the non-exclusive right to reproduce and distribute my thesis, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles or books).

. *UOM Verified Signature* ……05/08/2022……

M. D. T. Dulanjana Date

The above candidate has carried out research for the Masters thesis under my supervision. I confirm that the declaration made above by the student is true and correct.

…… *UOM Verified Signature* 05/08/2022 ……

Dr. Kutila Gunasekara Date

ABSTRACT

JavaScript is a prominent scripting language in web browsers. Earlier JavaScript was only used for client-side programming of browsers. JavaScript has been becoming a mainstream and a significant language in all types of large-scale applications from recent past.

JavaScript is an interpreted loosely typed language. JavaScript does not compile the source beforehand instead it translates codes at run time, line by line. JavaScript does not contain data types. Therefore, it is called a loosely typed language as well.

JavaScript does not consist fully pledged Object-Oriented concepts implemented. Instead, it uses prototypes to mimic the behavior of Object Orientation.

JavaScript was initially developed in 10 days by the founder Brendan Eich during the emergence of dot-com bubble and browser wars. At later points he himself had also admitted regarding some design mistakes of JavaScript.

Due to these reasons JavaScript has certain innate performance issues. To address these issues many researches have been done. Most of these researches have been focusing on fixing the performance issues of interpreting and loosely typed nature, by such solutions as JIT compilers and Typescript.

Hence this research will be focusing on an area which was given less significance, namely the coding standards. Research will first validate the common notion of coding standards have an impact on performance of JS and continue to provide an analytic document as a reference document of setting up a coding standards for any particular entity.

ACKNOWLEDGEMENTS

I would like to express profound gratitude to my advisors, Dr. Kutila Gunasekara and Dr. Malaka Walpola, for their invaluable support by providing relevant knowledge, materials, advice, supervision, and useful suggestions throughout this research work. Their expertise and continuous guidance enabled me to complete my work successfully.

Further I would like to thank all my fellow students who helped me on finding the relevant research material, sharing knowledge and experience and for their encouragement.

I am as ever, especially indebted to my parents for their love and support throughout my life. Finally, I wish to express my gratitude to all my colleagues at all my workplaces, for the support given to me to manage my MSc research work.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

API – Application Package Interface

CDN – Content Delivery Network

CSS – Cascading Style Sheets

DNS – Domain Name Server

HTML – Hyper Text Markup Language

JIT – Just in Time

JS – JavaScript

JSON – JavaScript Object Notation

SQL – Structured Query Language

CHAPTER 1

INTRODUCTION

1.1 Introduction to JavaScript

JavaScript [1] has a longer history than twenty years. Preliminary goal of JavaScript was to work as a client-side scripting language for web browsers. Until to the day JavaScript or JavaScript based libraries and frameworks are the principal way of frontend manipulation of all the web browsers.

With recent technological changes, JavaScript has changed its role from a scripting language for web browsers into a mainstream language to develop fully functional web, mobile, desktop or cross platform applications.

Scripting languages are not expected to perform much of arithmetic or logic operations and other manipulations. But when JavaScript becomes a mainstream programming language it is being used to perform stronger business and technological problems.

Apart from frontend libraries and frameworks JavaScript had been adopted to the server-side technologies and eventually had become a server-side language as well. Furthermore, JavaScript is now used to manipulate databases, especially when it comes to No-SQL databases JSON (JavaScript Object Notation) is also much popular.

1.2 Problem Identification

Since JavaScript emerged as a mainstream language, performance improvement of it became an essential research and development topic. There have been several new developments such as JIT, Typescript as performance improvement solutions. In addition to them there is another type of performance improvement that can be considered which is related to the standards of coding. It had been identified as a problem in the preliminary stages of this research that formal research and development of such a standardization is lacking in the current context of JS development. Since JavaScript has various functions available to perform the same

data manipulation, it is important to analyze and identify the better performing standards and produce a reference material so that all the commercial JS developers and JS framework and other tool developers can be benefitted equally.

1.3 Research Objectives

The objectives of this research are to:

- Address the lack of formal research in the area of coding standards related performance optimizations in JavaScript development.
- Compile a standard document as a reference for all sorts of JavaScript developers to optimize their performance tweaks and as an assistance to setting up a coding standard for their own context such as project or company.

1.4 Motivation

JavaScript has gained a considerable level of popularity in the industry level to be a mainstream language for large scale applications and JavaScript is now emerging as a modern area of research and developments are the main motivational reasons for the research.

Since the JavaScript is adopted as a mainstream language multiple libraries and frameworks have been developed and evolved using it. Most of these frameworks are web-based frameworks. And also there are some frameworks support for desktop or cross platform applications development as well as for database handling.

With these emerging use cases, JavaScript cannot only include mere characteristics and qualities of a scripting language as its initial developments. Rather it is expected to improve on the performance factor of the language. Due to this new requirement,

considerable amount of research and development have been done to improve the performance factor of JavaScript.

According to the Scripting language nature of JavaScript it has two major issues when it comes to large scale applications. First issue is that JavaScript is not a compiled language rather it is an interpreted language. Secondly JavaScript is a loosely typed language.

To address the interpreted nature of JavaScript, JIT (Just In Time) [2] compilers have been developed. JIT compilers do not compile the JavaScript applications beforehand. Instead JIT compiler analyze the code and identify the hotness of the code segments. Hotness is the measurement of significance and the amount of contribution of a certain code segment for the execution time. If a code segment is identified with a higher hotness JIT compiler compiles that specific code segment to avoid unnecessary recursive interpretations. To address the loosely typed nature deconstructing the type system have been researched.

With the new researches there are various tweaks available for improving performance of JavaScript. But when it comes to development same coding standard cannot be applied to all the business cases or especially logical cases. Therefore, it is beneficial for developers to have a system to identify which kind of coding standard best suited for their specific case. To compile such mechanism by empirically analyzing different coding standards of JavaScript is another motivational factor for this research. It is also notable that there's no any research done following proper research methodologies regarding this specific area. Hence it is also notable that it is essential to empirically prove the general conception of coding standards have a performance impact.

1.5 Summary of Research Findings

1.5.1 Coding standard changes

Changing the patterns and techniques as well as the common practices of coding can have a higher impact on the performance than expected. Main purpose of this research is also to find out those areas and identify the best practices for given scenarios.

A practical research approach has been taken towards this phenomenon. Very common funcition like eval() function or new keyword and really simple code snippets such as string concatenations are also a part of performance improvement.

Benchmarking has shown number of operation per second is considerably higher when using new keyword instead of eval() function.

Another interesting finding is about the impact of string concatenation in four methods.

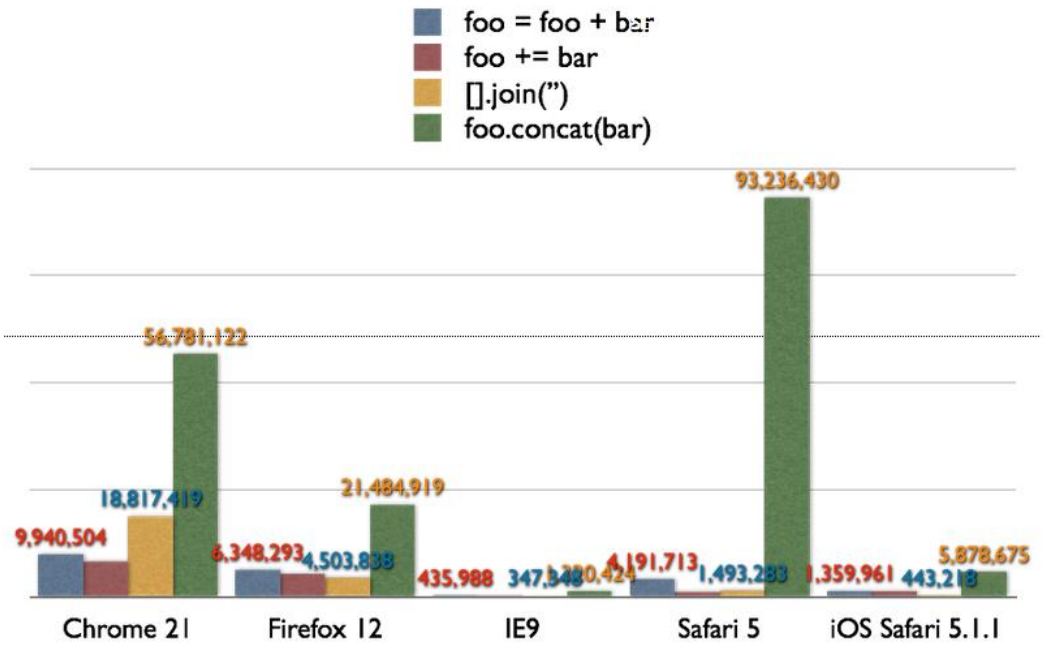

Figure 1. 1 Number of operations per second for string concatenations

According to the Figure 2 in all the browsers there is a significant performance improvement when 'concat' keyword is used instead of concatenation operator.

When using loops, it has been identified that it is better to use for loops instead of while loops when it can be. But in older browsers such as Internet Explorer 9 and earlier while loops were faster.

When accessing object values in deep level, for recursive accesses in same static path can be saved at first instead of accessing the same long path again. This simple change has a significant effect in all the browsers.

This performance analysis has been taken place upon commonly used code snippets in browsers. In the scenario it is considered only about web applications. But the same research can be advanced by adding logical scenario-oriented code snippets in cross platform nature.

Changing of coding standards will have an impact on the performance of JavaScript. These standards include client-side code changes as well as server level configurations.

These coding standards are mainly based on a single theory. HTTP requests takes more time than the time it takes to execute in the client side. Therefore, number of HTTP requests has a direct link to the performance of JavaScript since the execution is usually blocked by HTTP requests until they are completed. [3]

In a research regarding HTTP requests and caching it has been found that from the total web app loading time it is only taking around 9% for initial large HTML file loading and 91% is consumed by following multiple HTTP requests.

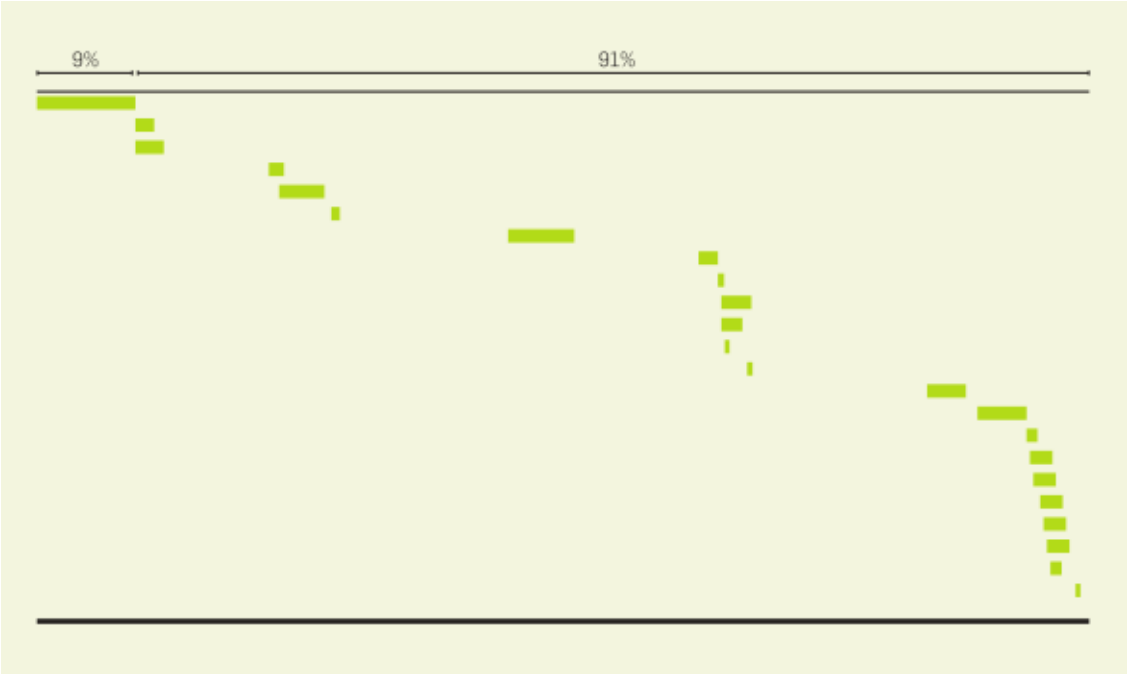

Figure 1. 2 iGoogle HTTP traffic with an empty cache.

When prime cache is used this following request time consuming is reduced to 83% of overall time.

Since the following HTTP requests are for fetching JavaScript, Cascading Style Sheet and image files, concatenation of multiple files into together, reduce size of text by minifying, reduce file sizes by compression are identified techniques to improve the performance.

Apart from minimizing the number of requests there is another approach to reduce requests by keeping cache.

Additionally there are other best practices such as using a Content Delivery Network (CDN), minimize Domain Name Server (DNS) lookups, avoid redirects, make JavaScript and CSS external files and import CSS at top of the HTML file while importing JavaScript files at the bottom of the HTML page are some of those identified best practices. [4]

1.5.2 Just in Time Compiling

Just In Time Compiling or JIT compiling is the way of compiling code while running. In JavaScript Just In Time compiling is used to improve performance by avoiding certain drawbacks of interpreting nature.

Interpreters are quick to get up and running. It doesn't have to go through that whole compilation step before it can start running the code. Interpreter just start translating the first line and running the code.

Because of this, an interpreter seems like a natural fit for something like JavaScript. It's important for a web application to be able to get going and run the code quickly as the user cannot be kept on waiting for compiling the code after receiving it. Such a scenario will further downgrade the performance or at least users will feel the processing is slow. That's why browsers used JavaScript interpreters in the beginning.

But the drawback of using an interpreter comes when the same code is executed more than once. For example, in a loop interpreter has to interpret the same instructions recursively for each iteration of the loop. It will obviously slow down the performance of the execution.

In contrast a compiler takes a little bit more time to start up because it has to go through that compilation step at the beginning. But then code in loops runs faster, because it doesn't need to repeat the translation for each pass through that loop.

Another difference is that the compiler has more time to look at the code and make edits to it so that it will run faster. These edits are called optimizations.

The interpreter is doing its work during runtime, so it can't take much time during the translation phase to figure out these optimizations.

JIT compilers have profilers. Profilers monitor the execution of code. At first it will run the code straightaway using interpreter. If any code segment executes few times, then it is marked as warm whereas if it runs several times, will be marked as hot. When the code gets warm JIT will send it off to compiling and after that compiled code will be used in the execution.

Even for the JIT researchers there's a major constraint that JavaScript has a dynamic nature. Dynamic nature is occurred due to loosely typed syntax of JavaScript. In a verbose manner we can say that data types of JavaScript variables change on the fly. Due to this nature there are researches on special type JIT compile called trace based JIT compilers.

The system starts running JavaScript in a fast-starting bytecode interpreter. As the program runs, the system identifies hot (frequently executed) bytecode sequences, records them, and compiles them to fast native code. Such a sequence of instructions is called a trace.

Unlike method-based dynamic compilers, these dynamic compilers operate at the granularity of individual loops. This design choice is based on the expectation that programs spend most of their time in hot loops. Even in dynamically typed languages, we expect hot loops to be mostly type-stable, meaning that the types of values are invariant. [5] For example, we would expect loop counters that start as integers to remain integers for all iterations. When both of these expectations hold, a trace-based compiler can cover the program execution with a small number of type-specialized, efficiently compiled traces [6].

1.5.3 Type Deconstructing

Type deconstructing is used to address the identified loosely typed condition. Trace based JIT compilers are also addressing the same issue. Type deconstructing is addressing the same issue but using a different methodology.

With recent evolvement of HTML5, JavaScript performance has been considered largely and it is proven to improve using many popular benchmarks such as Kraken [7], Octane [8], and SunSpider [9]. But a research has identified that these benchmark improvements are not applicable in most of the real time scenarios. This is found out by using a popular service called JSBench [10] which gathers real codes from real websites. For those codes popular JavaScript engines have been failed to display a significant performance improvement. According to the research it seems to be that, such a situation has been taken place due to the dynamic nature of JavaScript.

In the research they decouple the prototypes and method binding n JavaScript. Because changes in the prototypes or method bindings of otherwise identically structured objects resulted in type unpredictability. [10]

CHAPTER 2

LITERATURE REVIEW

2.1 Status of JavaScript Research

JavaScript is arguably becoming the most popular language all around the world. In the latest version of famous TIOBE index in January 2018, JavaScript has climbed up to the rank of 6 in popularity.

Even though the JavaScript is gaining popularity it has lots of negative comments and feedbacks by individual critics and professionals on various online forums. Negative comments are given mainly due to the poor design of it in the initial release in 1999 [11].

JavaScript was and is the prominent operator of web applications since almost the beginning of the web. From that niche position JavaScript has been evolving into a much significant position with the new technologies emerging for past few years.

With this escalation of role of JavaScript, it is becoming a prominent computer language in almost all of the platforms. These platforms include and not limited to web, desktop, mobile environments and Operating systems such as Windows, Linux, Mac and Android.

With the popularity of JavaScript, new JavaScript engines are developed by multiple vendors and communities. Google's Chrome V8 [12] and Mozilla's SpiderMonkey [13] are the prevalent prominent JavaScript engines.

Because of this extreme pervasiveness and unavoidable requirement to be executed on various platforms and numerous devices, optimizing JavaScript code and improving performance of JavaScript applications has become a real challenge.

2.2 JavaScript Engines

JavaScript Engines are programs which were initially meant to be Simple JavaScript interpreters. Later with the heavy use of JavaScript in web applications vendors and communities have started to improve the quality of JavaScript engines. As a result features like JIT compilation of JavaScript code before interpreted are added to the JavaScript engines.

First JavaScript Engine code named SpiderMonkey was developed by Brendan Eich [14], [15] who is considered as the father of JavaScript at Netscape Communications Corporation for the Netscape Navigator web browser. Since then number of JavaScript engines were developed for various browsers and purposes.

When consider the performance of web browser as a whole rather than JavaScipt engine, layout engine also has a vital effect. Layout engine is an application which is used to render web pages in the context of web browsers. In a broad definition layout engines are application which combines content with formatting to create any kind of structure.

		Engines	
Browser	Manufacturer	Layout	JavaScript
Firefox	Mozilla	Gecko	SpiderMonkey
Chrome	Google	Blink	V8
Internet Explorer	Microsoft	Trident	Chakra [16]
Edge	Microsoft	EdgeHTML	ChakraCore
Opera	Opera	Blink	V8
Safari	Apple	WebKit	Nitro

Table 2.1 Details of JavaScript engines

2.3 Analyzing JavaScript

Performance of JavaScript applications has become a key concern in JavaScript application development researches. We can identify mainly 2 types of JavaScript code and app analysis to identify potential segments of performance degradations.

2.3.1 Dynamic Analysis

Dynamic Analysis is the way of analyzing application source codes by executing them. Dynamic analysis gives a more practical approach to find the performance of a segment of code on a given platform.

In the case of JavaScript most of the researchers have focused on dynamic analysis to find the bugs and optimize the code.

Dynamic Analysis researches are more popular with JavaScript. Principal reason for that popularity is the behavior of JavaScript language. JavaScript is mainly an event driven language. In layman's terms mostly used as a client-side language JavaScript is usually invoked by action of users and the flow is interrupted by the events created by users. Since JavaScript has comparatively low static flow of source dynamic analysis is preferred.

2.3.2 Static Analysis

Static Analysis is the way of optimizing the code without executing the code firsthand. Static Analysis can be done on Source Code, Byte Code or machine code. Static analysis is usually done using automated tools. IDEs showcase similar behavior of this kind of tools. They are helpful for the developer to understand the bugs at development time. Sophisticated versions of these tools are supporting for high level of analyzing and optimization of codes for better performance. Humans can also analyze the source code without executing. Usually it is referred as program understanding, program comprehension or code review.

There are some static analysis methods have been developed for JavaScript which

mainly covers the areas of type checking [] or security[]. Static analysis has methodologies which resembles the methodologies of benchmarks which are elaborated in section 2.4.

In the case of JavaScript there are very little amount of static analysis research are done when compared to researches done on other programming language paradigms.

2.4 Benchmarks

Benchmarks in general can be defined as applications which allows to compare things against a well-defined goal about any certain feature. Benchmarks are not only limited to Software. It is heavily used in hardware segment to compare performance of hardware such as CPU, mobile phones, etc.

Benchmarks are usually analyze sequential flow of certain tasks which are not interrupted while executing. After analyzing it provides one more scores related to the various measurements of performance. These scores are used to compare the respective hardware or software system.

In the context of JavaScript there are 2 types of Benchmarks available. JavaScript was initially intended to use as the client side scripting of browser applications or simply web pages. Lately with the improvements of JavaScript engines it has come out of browsers. Because of that 2 benchmark categories analyze the performance of JavaScript as a part of browser and other category checks the performance of JavaScript execution only. Former category includes performance of JavaScript execution along with Web page rendering and DOM (Document Object Model) manipulations.

Other type are pure benchmarks on JavaScript Manipulation. Some of these benchmarks are developed by the developers of JavaScript engines themselves to measure the performance of their projects.

Table 2.1 [17] shows the overview of Existing JavaScript benchmark details. Column named ‘Full’ is denoting whether the particular benchmark is benchmarking the full

browser or only the performance of JavaScript execution. Column named 'JS engine' defines whether it is developed by a JavaScript engine developer, if so the name of the developer.

Name	Developer	JS engine	Full
Benchmark.js	Mathias Bynens, John-David Dalton	-	No
Browsermark	Rightware	-	Yes
Dromaeo	Mozilla	SpiderMonkey	No
JSLitmus	Robert Kieffer	-	No
Kraken	Mozilla	SpiderMonkey	No
Octane	Google	V8	No
Peacekeeper	FutureMark	-	Yes
SunSpider	WebKit	WebKit	No

Table 2.2 Details of JavaScript Benchmarks

Apart from benchmarking certain sequential code segments, researches [17] have showcased that benchmark scores are not accurate for the real world applications. Benchmarks usually use JIT compilation and use less amount of anonymous functions. These decisions may lead to different behaviors of real time applications' performance when it compares to the way they behave in a benchmark.

Additionally benchmarks can only analyze sequential flow behavior of a source code. Hence it is not much suitable for event driven programming language like JavaScript. Therefore a better method to analyze JavaScript performance would be profilers.

2.5 Profilers

Profilers [18] are the applications or toolset which are useful for analyze dynamic behavior of applications. Unlike benchmarks profilers do not expect sequential uninterrupted code segments. Therefore, profilers are much suitable for event driven languages like JavaScript. Profilers monitor for the usage of CPU and Memory, execution time, etc. during the execution of dynamic programs. Profiling is also much popular to use at development time to identify bugs in the code or depletion of performance in the code before release.

Profiling enables to find out which functions are heavy on using CPU time or memory use. With this insight to the functions developers can optimize the particular function or the application as a whole to achieve better performance and reduce unwanted memory usages as well as memory and disk accesses.

In the general case browsers contains developer tools along with profilers by default. These profilers are usually capable of measuring CPU time as well as memory usage of JavaScript executions. Apart from the in-built tools browsers support external tools as well. Firebug is such a popular external tool which have been lately adopted to the internal Firefox developer tools.

2.6 Performance

Performance is crucial for JavaScript applications. Mostly JavaScript is presenting as a client side language. End users are much concerned about the performance of applications as they use it. It can be either in the case of browser application or standalone JavaScript applications.

Performance can be identified as a general term. When performance is considered as an individual entity it has less empirical value. Performance of something is usually perceived as a qualitative value. In order make it quantitative or empirical as we need for this research as well, we have consider measurable units of performance. These units do not measure performance as a whole. Instead they measure various aspects which are considered as factors of performance.

Performance measuring factors are not independent units. They are dependent on the context. These measures needs to be identified and defined according to the nature of application and the goal of performance evaluation. Generally used performance measuring factors are explained further.

2.6.1 Execution time

This is a primary measurement factor. This is used to measure the time it takes to execute part of a code or a whole application. Elapsed time can be measured in either milliseconds or ticks of CPU clock.

2.6.2. Throughput

Generally, throughput measure the amount of any relevant thing that has been consumed within a unit time. Amount of lines executed, I/O operations, etc. can be considered for calculating throughput.

2.6.3 Resource Consumption

Resource consumption is straight forward measurement. It measures the hardware utilization throughout execution. Maximum and optimal utilization is considered as better performing execution.

2.6.4 Response Time

This measurement has 2 aspects. In end user perspective response time is one critical measurement factor. End users perceive this factor as more into qualitative way rather than a quantitative manner.

Response time is usually an important factor for network applications. Response time is the time between a request initiation and the time to receive the response. Hence this is an important performance measure in web applications. In the context of standalone applications response time is equals to the execution time.

2.6.5 Bandwidth

Bandwidth defines the speed of communication between 2 or more components. Similar to the response time, term bandwidth is highly used with network applications. It defines the amount of data flow between networked devices in bits per second.

For the internal or single unit communication bandwidth mainly refer to the data flow between internal components such as from CPU to Memory.

2.7 Properties of Performance evaluation methods

Since there are multiple methods of performance evaluation, it is important to define and quantitative way to identify suitable analyzing techniques for any particular application. In order to compare the analyzing techniques it is important to identify the properties of analyzing methods.

2.7.1 Accuracy

Accuracy is the most important factor of any analyzing method. Analyzing scores should be accurate to conclude the optimization decisions. Incorrect analyzing scores will lead to incorrect optimization decisions. Also it should enable the developers to correctly identify the location of performance issues.

2.7.2 Impact

When an application is analyzed, analyzing method should not have a significant impact on the performance of the application. Otherwise the comparison of analyzing methods will not be accurate and will be biased. If the analyzing methods have any impact to the performance it should not be a significantly large difference. Or else the impact should be constant and that constant value should be removed from the execution time.

2.7.3 Usability

When performance is analyzed not only the developer is involved, but also the administrators and end users are involved. Therefore these analyzing methodologies should be usable with low learning curve. In the case of end users their intended work should not be interrupted or unwanted additional work shouldn't be added.

2.7.4 Portability

These methods should easily be available to port into other projects. In the context of a commercial organization reusability is cost efficient method. Therefore vendors prefer portable methodologies to analyze performance.

2.8 JavaScript Frameworks

With the popularity of JavaScript multiple frameworks are being developed for various purposes. Almost all the JavaScript applications are developed using some sort of one or more frameworks rather than using vanilla JS. Therefore for a complete analysis of performance it is essential to consider the performance of frameworks as well. Since there are numerous amount of frameworks have been developing, best way is to filter out the highest popular frameworks out of them.

Main cause for analyze frameworks separately is that frameworks use their own design principles and their own standards. Due to various reasons these standards may leads to sort of code smells. Code smells are not essentially bugs of a code but the parts where it can lead to future issues. A code smell is a surface indication that usually corresponds to a deeper problem in the system [19]. Popular controversial article named 'you have ruined javascript' [20] which accused AngularJS developers for smelly code is a good example of arguable situation of performance of JavaScript Frameworks.

2.9 Root Causes for Performance Degradation

Researches related to JavaScript coding standards issues have showcased that the

performance issues can be categorized into certain groups of main reasons or root causes.

Previous research [21] has identified a proper methodology to find out causes of optimization degradation in multiple open source JavaScript projects.

Project	Description	Kind of platform	Total LOC	Number of issues
Angular.js	MVC framework	client	7,608	27
jQuery	Feature-rich library	client	6,348	9
Ember.js	MVC framework	client	21,108	11
React	A declarative library	client	10,552	5
Underscore	Utility library	client and server	1,110	12
Underscore.string	String manipulation helper	client and server	901	3
Backbone	MVC framework	client and server	1,131	5
EJS	Embedded templates	client and server	354	3
Moment	Date manipulation library	client and server	2,359	3
NodeLruCache	Caching support library	client and server	221	1
Q	Library for asynchronous promises	client and server	1,223	1
Cheerio	jQuery implementation for server	server	1,268	9
Chalk	Terminal string styling library	server	78	3
Mocha	Test framework	server	7,843	2
Request	HTTP request client	server	1,144	2
Socket.io	Realtime application framework	server	703	2
Total			63,951	98

Figure 2.1 Identified issues in JavaScript projects

Figure 2.1 shows the count of identified issues from JavaScript projects. For the evaluation both front end and backend frameworks have been considered. But vanilla JS has not been considered for the evaluation. All the selected frameworks are open source projects. That decision was taken since finding smelling codes in closed source projects are not possible. Also the popularity of the projects is considered such as number of pull requests in GitHub. Popularity concern is important when it comes to importance and usage of the research as well as for the increased chances of having performance related issues.

All the performance related issues have been found out from GitHub repositories by analyzing issue reports. Identification has been done on either inspecting obvious labels or otherwise by searching for performance related keywords such as “performance”, “optimizations”, “responsive”, etc. In this particular research, solutions are not generated instead solved performance issues have been considered and they are confirmed via executing test cases to analyze the improvement and reproducibility.

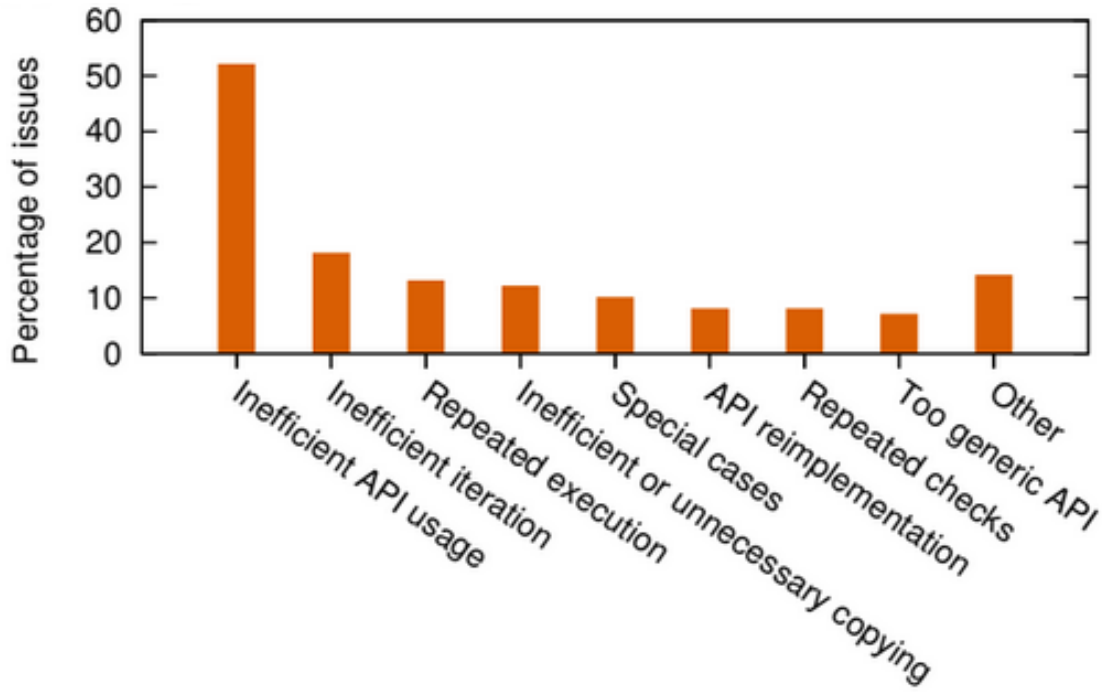

Figure 2. 2 Principal Root causes

Figure 2.2 shows the currently identified root causes of the above mentioned research. It is obvious that most of the performance issues occur due to the bad coding practices. Largest value indicates that even though developers are aware and use many APIs to improve the efficiency and productivity, yet they're less knowledgeable of the best practices on how to use them appropriately.

Figure 2.2 shows the various APIs that have been used inefficiently. Most of the cases are related to reflection and usage of strings. Optimizations done by community developers show that most of the cases developers were only concerned on the functional implementation and nonfunctional details are evaded.

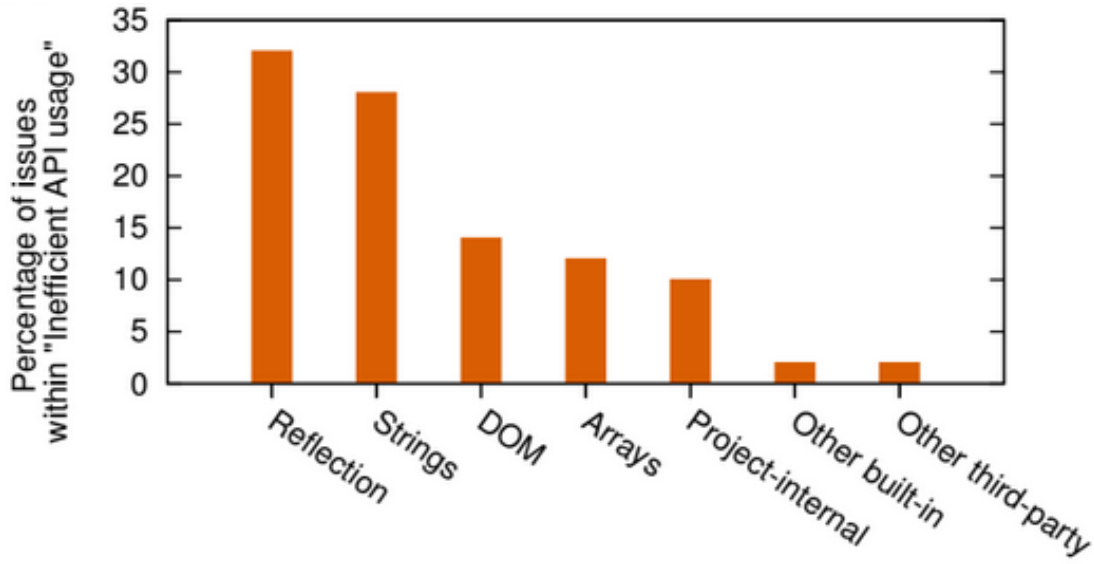

Figure 2. 3 Count of issue types in API usage

CHAPTER 3

METHODOLOGY

3.1 Overview of the Prospective Methodology

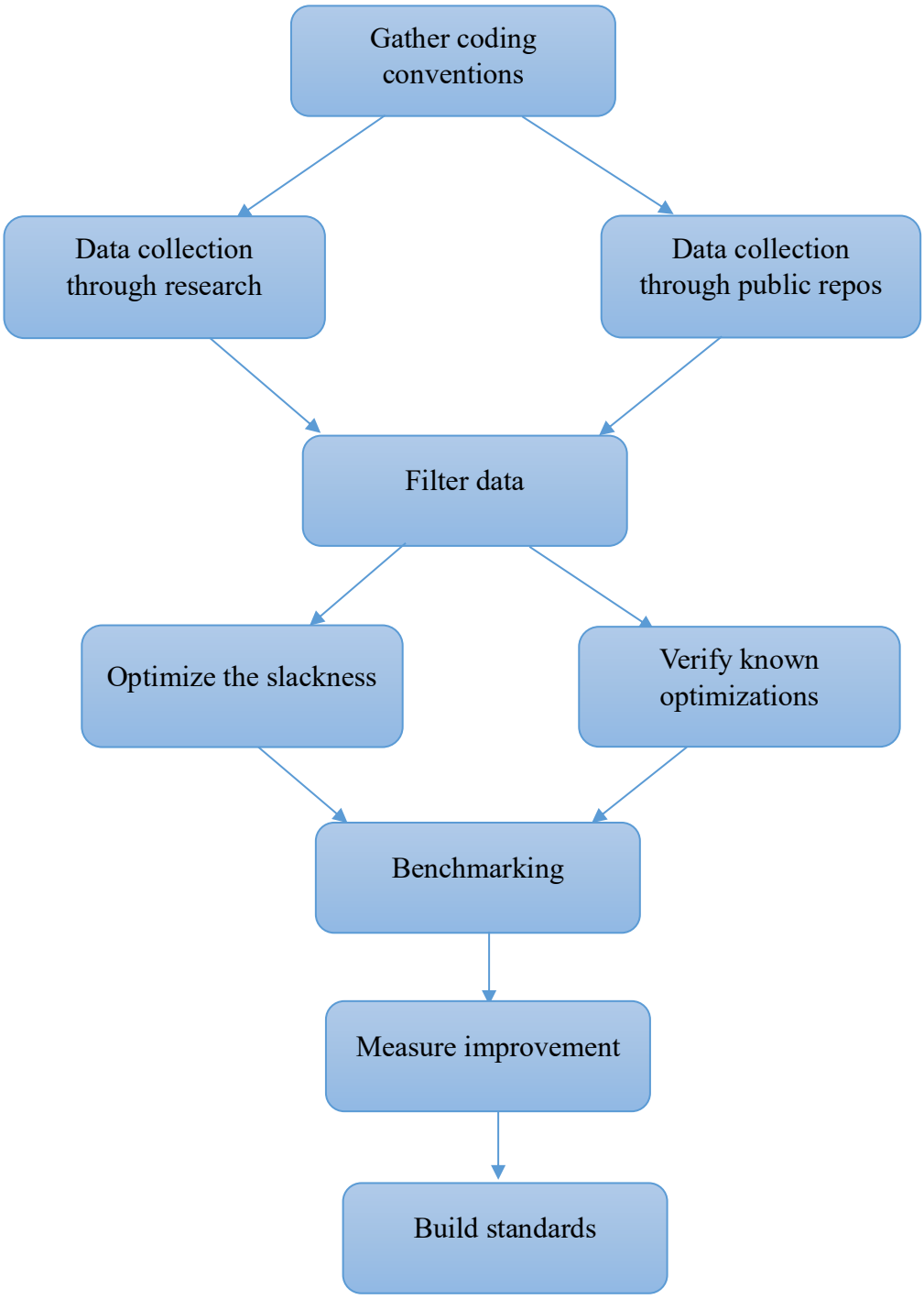

Figure 3. 1 Overview of suggested methodology

3.2 Identifying coding standards

Initial stage of the research had been solely focused on gathering necessary data to analyze. Major part of the gathering is based on publicly available resources varying from researches to web articles. In this sub phase currently known or discussed and suggested optimizations will be identified through public sources. Solutions for the current performance issues will also be developed as required to proceed. In this sub phase main concentration will be kept on vanilla JS.

Data from publicly available open source repositories will also be gathered in parallel. In this segment identified methods from previous researches will be used. In this sub phase server side as well as client side frameworks will be considered. In the selection process popularity will be considered as the criteria. Popularity will be defined by criteria based on online polls, GitHub pull requests and issue reports. To identify the performance issues of these projects, same methodology will be used. It will either be recognized using a dedicated flag or searching for performance related keywords.

Gathered information needs to be verified before the analyzing phase. For this phase test cases will be developed to reproduce the issue in original code and another relevant test cases to verify the optimization. These test cases will be newly developed ones or reuse the given test scenarios in issue reports or in the repository.

3.3 Analyzing Optimizations

After gathering of data it will be filtered for the relevancy as well as to remove the duplicates. Furthermore, removal of optimizations which are already firmly confirmed will be tried as much as possible.

Analysis is based on benchmarking rather than profiling. Profiling is more focusing on whole operation of a single entity to identify the cycles of operations or hotness. Benchmarks are a more relevant method to analyze the performance of same entity on different environments.

After the benchmarking, measure results of improvements will be mainly presented in graphical format. Important types of graphical representations of improvements have

been identified using previous researches.

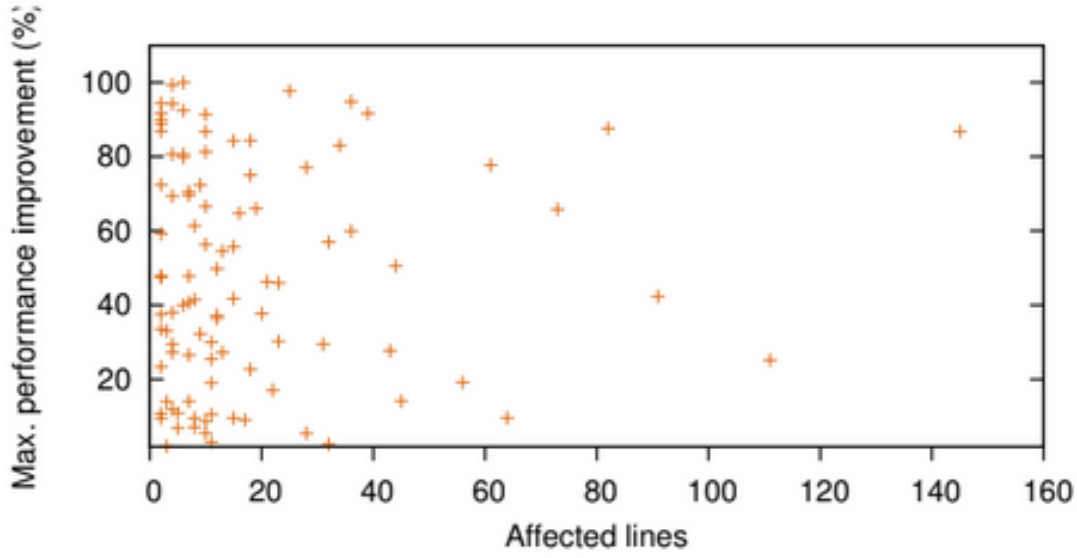

Figure 3. 2 Improved performances against the effort

Figure 3.2 showcase a very important aspect of how to use the standards set by this research. It denotes the refactoring of code to achieve a certain performance level needs to be determined based on the time and effort that we spend on it. This specific scenario is not captured within the scope of research as it is not possible to grasp this idea with micro benchmarks. Nevertheless, this factor is noteworthy as a deciding factor of a performance refactoring in real world applications.

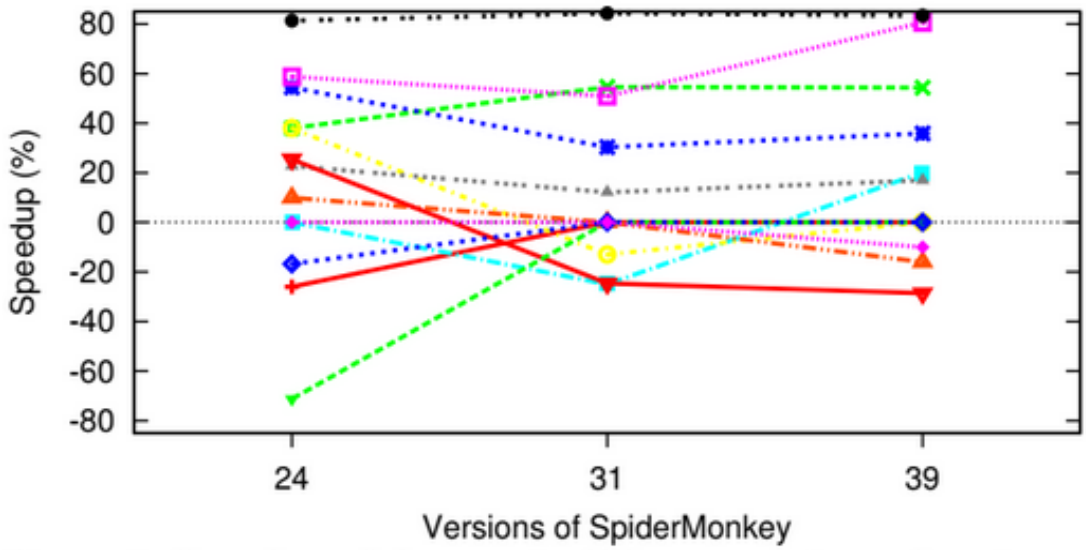

Figure 3. 3 Speed variation with SpiderMonkey versions

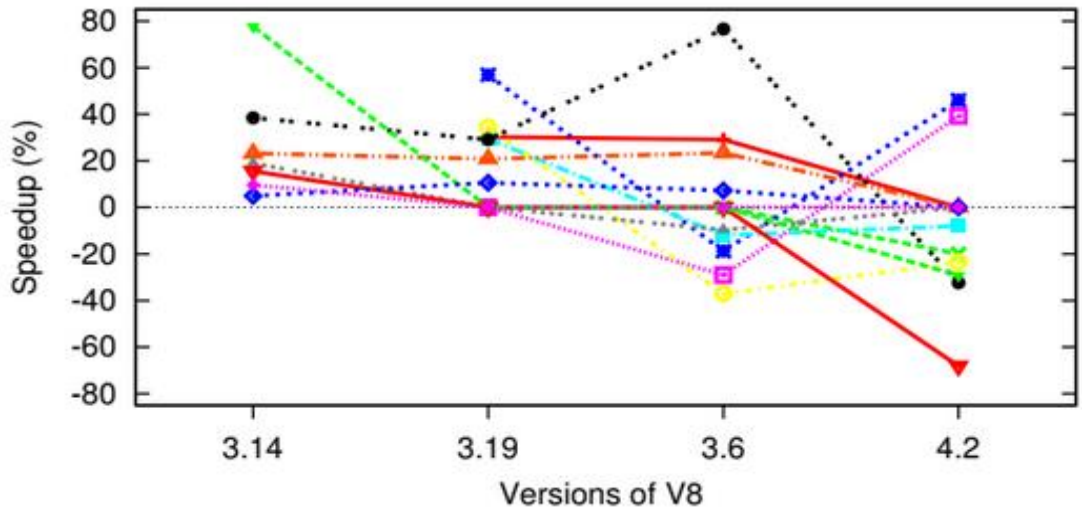

Figure 3. 4 Speedup variation with V8 versions

It is important to identify the impact of optimization in a cross browser environment. As some researches have already revealed some optimizations may actually end up as degrading the performance in certain versions of JavaScript engines. Figure 3.3 and Figure 3.4 are sample graphical representations presenting the level of speedup within multiple versions of same JS engine.

3.4 Building Standards

Final outcome of this research will be the rules to standardize the use of optimizations in various JavaScript engines. In a more elaborative manner this will showcase the impact of already identified improvements as well new improvements in various platforms and how to use them optimal way.

Further if the time permits cross engine speedup levels can be calculated to identify the suitable JS engines for a given particular task.

CHAPTER 4

IMPLEMENTATION

4.1 Introduction

This chapter describes the implementation details of the methodology explained in the methodology section and the outcome of the implementation. Research scope limits the performance study to one specific programming language. For the benchmarking purposes developing a brand-new benchmark from the scratch is not covered under the scope of this research and performance analysis is done using the existing popular benchmarking tools.

4.2 Benchmarking

Benchmarking allows to compare the performance of different code snippets. For the purpose of research 3 popular benchmarks have been identified.

http://jsbench.github.io/

https://jsben.ch/

https://www.measurethat.net/

Theoretically all the JS benchmarking tools behave in a similar way. Difference between each benchmark tool is the design of inputting data and the format and style that the output is presented.

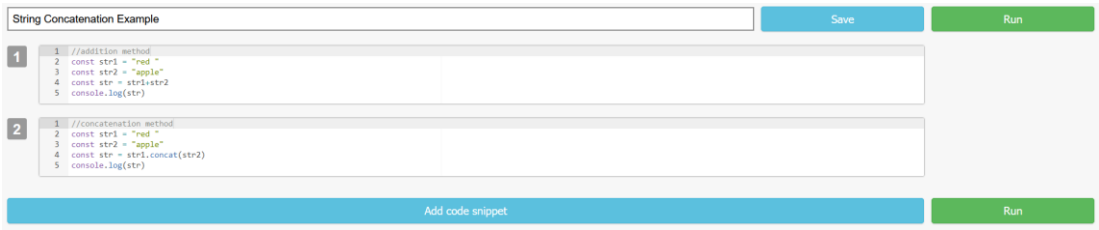

Figure 4.1:Input screen of jsbench.github.io

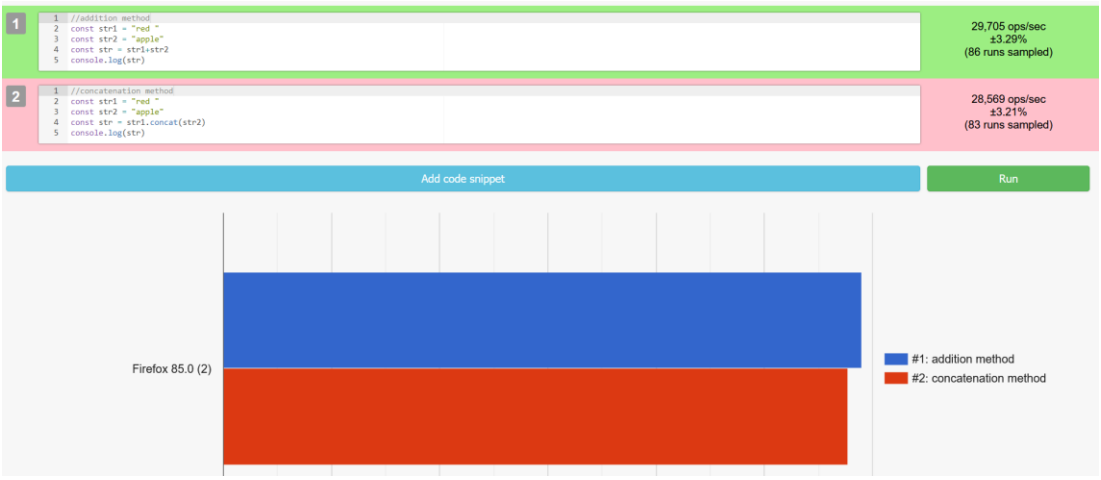

Figure 4.2: Output screen of jsbench.github.io

Figure 4.3: Input screen of jsben.ch

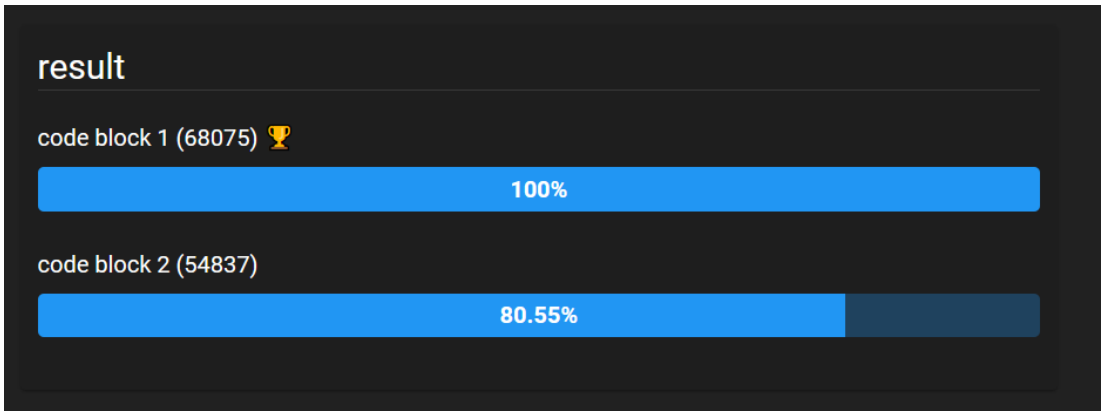

Figure 4.4: Output screen of jsben.ch

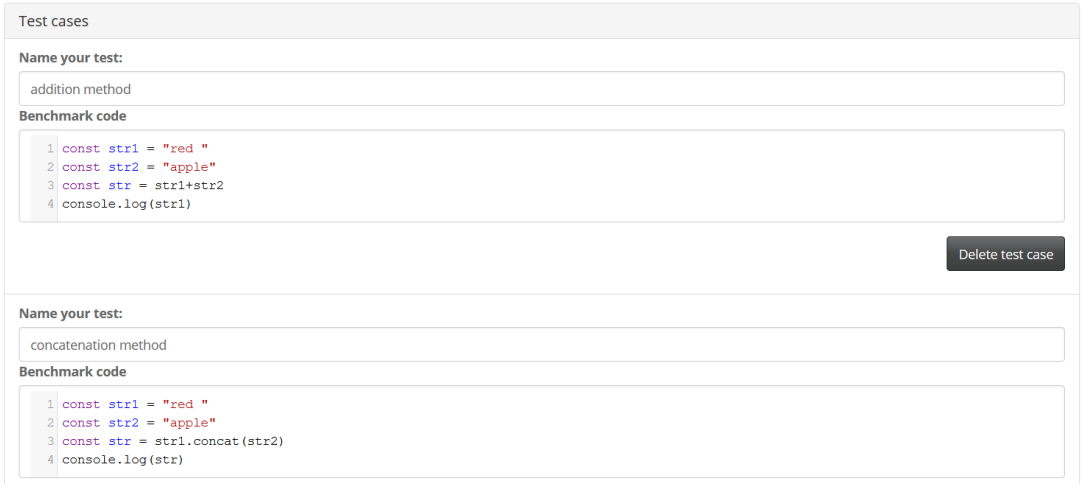

Figure 4.5: Input screen of measurethat.net

Test case name	Result
addition method	addition method x 24,613 ops/sec ±3.95% (48 runs sampled)
concatenation method	concatenation method x 23,528 ops/sec ±5.25% (49 runs sampled)

Fastest: addition method,concatenation method
Slowest: concatenation method,addition method

Figure 4.6: Output results screen of measurethat.net

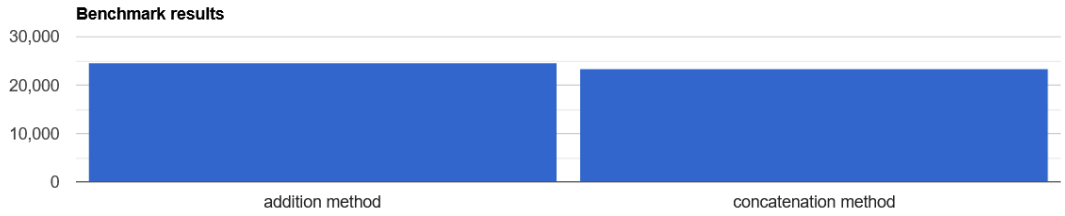

Figure 4.7: Output chart screen of measurethat.net

1.5 Understanding and avoiding common mistakes in Performance Analysis

1. Ensure Same amount of work is performed in each operation

   Usually in a modern computer simple JavaScript operation takes less amount of time than the clock time of the computer. Therefore, in order to capture the operation time, benchmarks repeat the same operation for N times and then divide the full amount of time by the number of operations which will return the time for a single operation (T).

   $$T = \frac{T\ X\ N}{N}$$

   Hence it is needed to ensure that the same amount of work has been performed in each operation through the N times.

   As an example, if a slice operation executed against a word with 10 characters, there won't be an operation to be executed after the 10th time. Henceforth all the operations of that code snippet will have 0 operational time, which will result an incorrect performance result.

2. JIT compilers of JavaScript engines are optimizing code up to a high extent, therefore the micro benchmark codes need to be looked for certain JIT compiler optimizations. Otherwise a benchmark may provide a false performance result. Compilers optimize out code in multiple disciplines, such as below mentioned optimization techniques.
   - Constant folding – JIT compiler identifies the calculation of constants and assign the calculated static value before the execution.
   - Constant propagation - Constant propagation assigns a static value to relevant calculations rather than reading from a variable location in each execution. Such that a constant reading in a micro benchmark is optimized.
   - Loop invariant – If a code segment is not changed within the execution of a loop, that code segment can be moved out from the loop.
   - Unused variables – If a variable is only assigned a value within the

execution and is not used in a subsequent execution, that variable will be eliminated.

- Dead code elimination – Some JIT compilers has the ability to eliminate dead code. As an example, if a JIT compiler optimize out a code segment within a loop, then the empty bodied loop is also eliminated as dead code.

4.4 System Environment

For the research, multiple environments are used to compare the performance in different platforms. Hence the developers can be more benefited as they can decide best coding standard specifically according to the deployment and client platforms of a project.

This research has used a mid-range hardware environment with Windows and Linux environments with multiple JavaScript engines. Nevertheless, the hardware platforms are unvaried to all the Operating System and JS engines. Hardware Details are as below.

CPU: AMD 4Core A10-8700P 3.2GHz
RAM: 8GB
Storage: SSD 256GB

Operating Systems and Browser Details are as below.

Windows 10:

Mozilla Firefox 85
Microsoft Edge 42
Google Chrome 87

Ubuntu 20.04 (Linux Kernel 5.4):

Mozilla Firefox 85
Google Chrome 87

In this research, performance impact of hardware such as CPU and Memory differences are not covered. Hence it had been determined to keep an unvarying hardware platforms to minimize the impact from hardware on the research results. In

the scope of the research primary focus was given to software platforms.

As a future research this could be expanded to cover more platforms which have different hardware standards and varying performance levels.

4.5 Output of the implementation

Expected implementation of the research will be the standardization documentation which elaborates the ways of implementing various logical and problem-solving tasks in JavaScript in the most optimal coding standard.

Target audience of the document which would be compiled as the output will be commercial JS developers, framework and other JS related tool developers. Different sectors of these developers will have different performance requirements. As an example commercial JS applications can determine a target platform where most of its users are relying on. But for general tool developers it is not possible to determine which platforms will be used by their respective clients or users. Therefore, they will have to optimize the performance in a general way. Due to that reason, evaluation will showcase the average performance gain and rankings along with individual performance comparisons among multiple platforms.

Standardization document will be divided into multiple sections based on the operations that could cause the performance degradation which have been identified during literature review phase.

- Variable handling
- String operations
- Array operations
- Object handling
- Iterations
- Asynchronous functions
- Other special operations

This documentation ideally should be a community-based web implementation of a documentation which continuously expands over time and may not be covered under the scope of this research.

And this documentation will include the comparison between multiple JavaScript engines in Windows and Linux platforms. For this research, hardware details are kept invariable.

CHAPTER 5

EVALUATION

5.1 Chapter Overview

This section discusses on the findings specifically on the performance details of using different coding standard approaches in order to solve the same programmatical or logical scenario. Evaluation will be segmented into seven segments as identified in the previous chapters on a logical basis of main JavaScript operation classes.

Also, the evaluation will consider the impact of hardware variations on performance standardization document. Evaluation will identify the factors on improving the document that have been disclosed during the research which will be discussed further in next chapter.

5.2 Evaluation process

Initially the process benchmarks are identified as common programing and logical scenarios that any ordinary JS developer comes across while developing a JS based application. Then for each of these identified scenarios, all the possible code solutions are identified. Then each of these coding solutions are run and get the operations per second value of each of them.

These benchmarks are basically executed in Firefox, Chrome and Edge browsers which comprise the large proportion of browser market share. As common to any experiment there is an error margin included in these benchmark results. In order to reduce the error of the results, every benchmark is executed for 5 times on each JavaScript engine and used the average value of the 5 times in evaluation process.

In the evaluation data sets, browser versions are used instead of JS engine versions due to 2 reasons.

Browser version is the more obvious versioning value and it is used in most of the questions and answers in online forums rather than the JS engine versions.

Browser version and JS engine has a one-to-one relationship. Therefore, if user needs to find performance details related to JS engine version for any special circumstance it is not difficult to map that value as well.

5.3 Impact of platform variance on Standardization document

Evaluation process has considered the impact of hardware and software variance of platforms on the performance and optimizations of JavaScript engines. A single benchmark is used to evaluate the performance impact.

Benchmark that has been used is a very common user scenario where a for loop is executed to do a simple variable defining and initialization. Yet the benchmark is facilitated with a very large array to reduce the number of operations per second, in order to distinguish the variations clearly rather than having a very large number of operations per second.

Method 1: Using native for loop without caching the condition

Method 2: Using native cached for loop

Method 3: Using for…of statement to generate a loop iterating over objects

5.3.1 Performance impact by various hardware capacities

In order to evaluate this observation, a benchmark is executed in same JS engines with different hardware capabilities.

	Lower Specifications (L)	Higher Specifications (H)
CPU	AMD 4Core A10-8700P 3.2GHz	Intel Core i5-7200U (7th Gen) 2.5GHz
Memory	8 GB	16 GB
Disk	SSD 256GB	SSD 512GB

Table 5.1: Higher and Lower hardware specifications

In the below given data set L denotes the lower hardware capabilities while H denotes higher hardware capabilities.

	Method 1	Method 2	Method 3
Firefox 89 (H)	665.449	649.129	147.16
Chrome 91 (H)	1273.827	1247.376	829.723
Edge (H)	1594.427	1621.03	1084.648
Firefox 89 (L)	206.931	197.227	59.549
Chrome 91 (L)	338.078	331.719	170.446
Edge (L)	351.413	350.432	173.133
Average	738.35	732.82	410.78
Rank	1	2	3

Table 5.2: Results depicting impact of hardware platform difference on javascript performance

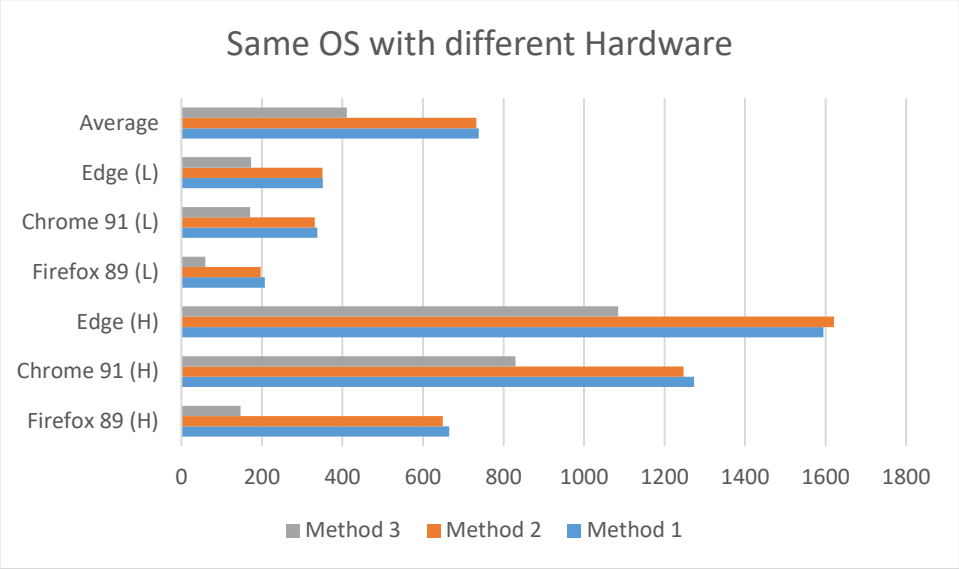

Figure 5.1: Chart of results depicting impact of hardware platform difference on javascript performance

By evaluating the data set we can deduce that different hardware performance has a vivid impact on the number of operations per second. Yet the proportional impact of hardware performance seems to be almost same or negligible.

In the provided benchmark chrome and Firefox have clearly followed the same order of performance results while the Edge browser has showcased a small variance. Even that variance is almost negligible since the number of operations per second value of method1 and method2 has a tiny difference. By considering the performance results of

lower hardware performance category winner can be declared as both method1 and method2. According to the above evaluation, it seems to be that hardware performance variations have less impact on the code level performance optimizations of JavaScript.

But still it is not feasible to rule out the hardware impact from the analysis when it comes to performance optimizations. Yet the inclusion of hardware variance will produce numerous amount of platform combinations, for the scope of the research it is not considered. Also as JavaScript is mainly executed at client side, number of hardware variances would not be technically feasible to addressed to. This should be considered and form a methodology in future work.

5.3.2 Performance impact by various Operating Systems

In order to evaluate this observation, same benchmark was executed in same JS engines running in different operating systems but with exactly same hardware. Test environment was a computer with dual boot facility for Windows and Linux operating systems. In the given data set, L denotes Linux and W denotes Windows Operating Systems, respectively.

	Method 1	Method 2	Method 3
Firefox 89 (W)	206.931	197.227	59.549
Firefox 89 (L)	128.118	122.404	39.267
Chrome 91 (W)	338.078	331.719	170.446
Chrome 91 (L)	346.041	347.375	187.732
Average	254.79	249.68	114.25
Rank	1	2	3

Table 5.3: Results depicting impact of operating system difference on JavaScript performance

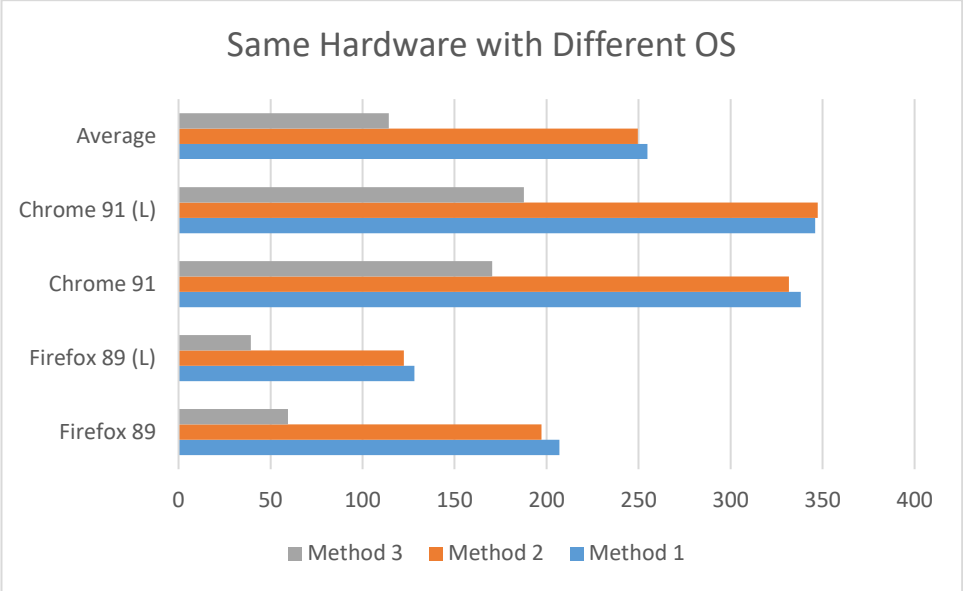

Figure5.2: Chart of results depicting impact of operating system difference on JavaScript performance

In both the test environments, performance results are proportionally identical. Performance results of same JavaScript engine in different Operating Systems showcase some quantitative differences but the proportional difference is almost negligible. Method1 and Method2 has a tiny difference in Chrome browser in Linux and Windows. But that difference can be ignored since it is negligible with respect to the number of operations per second value in the output.

After evaluating the above data set, it can be concluded that the performance results are almost identical to the results of same benchmark executed on different hardware performance levels. Henceforth it is established that similar to the impact of hardware variance, impact of Operating System variance is also negligible within the context of this research evaluation.

5.4 Performance Test Results

Performance test results are generated and collected in the measurement unit of operation for second values for each benchmark using different platforms. Additionally, a bar chart is drafted to assist the analysis process to conveniently compare the test results visually.

5.4.1 Variable Handling

var, let and const are valid variable defining ways in JavaScript. JavaScript variables can be defined in multiple ways. These different types of definitions determine the scope of variables.

5.4.1.1 Usage of var and let

Method 1: Using var

```
var arr = []
var a;
for (var i=0; i<100; i++) {
  arr.push(i);
}
for(var i=0; i < arr.length; i++) {
   a = arr[i]
   a = a+5
   arr[i] = a
}
```

Method 2: Using let

```
var arr = []
let a;
for (var i=0; i<100; i++) {
  arr.push(i);
}

for(let i=0; i<arr.length; i++) {
    a = arr[i]
    a = a+5
    arr[i] = a
}
```

	Method 1	Method 2
Firefox 89	2,453,125.60	2,627,733.95
Chrome 91	4,780,305.91	4,728,705.31
Edge	4,814,324.81	4,859,742.76
Firefox 89 (L)	1,700,062.70	1,710,705.17
Chrome 91 (L)	4,141,144.46	4,160,897.08
Average	3,577,792.70	3,617,556.85
Rank	2	1

Table 5.4: Results set, usage of var and let

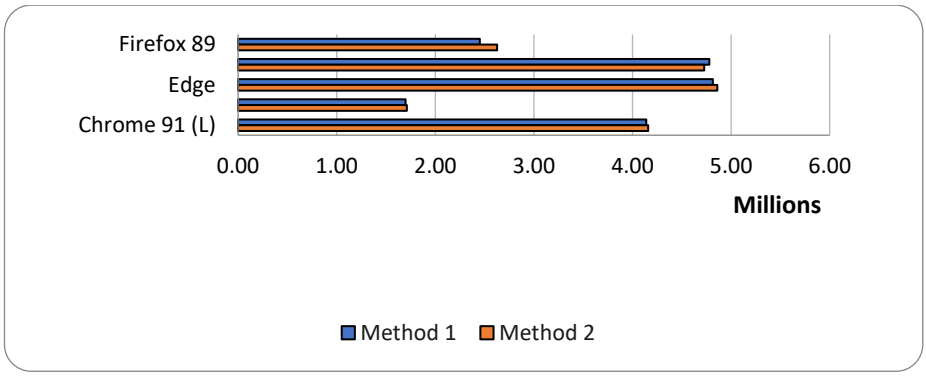

Figure 5.3: Results chart of var and let

5.4.2 String Operations

5.4.2.1 String concatenation

Method 1: Using plus operator

```
const str1 = "red "
const str2 = "apple"
const str = str1+str2
```

Method 2: Using template literals

```
const str1 = "red "
const str2 = "apple"
const str = `${str1} ${str2} `
```

Method 3: Using concat function

```
const str1 = "red "
const str2 = "apple"
const str = str1.concat(str2)
```

	Method 1	Method 2	Method 3
Firefox 89	446,867,226.18	487,037,074.94	27,829,354.81
Chrome 91	483,244,412.67	477,765,226.32	487,716,717.22
Edge	527,259,792.00	520,418,727.84	519,833,093.12
Firefox 89 (L)	441,850,469.92	437,118,689.71	17,836,481.49
Chrome 91 (L)	552,448,613.08	552,356,656.80	554,642,494.78
Average	490,334,102.77	494,939,275.12	321,571,628.28
Rank	2	1	3

Table 5.5: Results set, usage of String concatenation

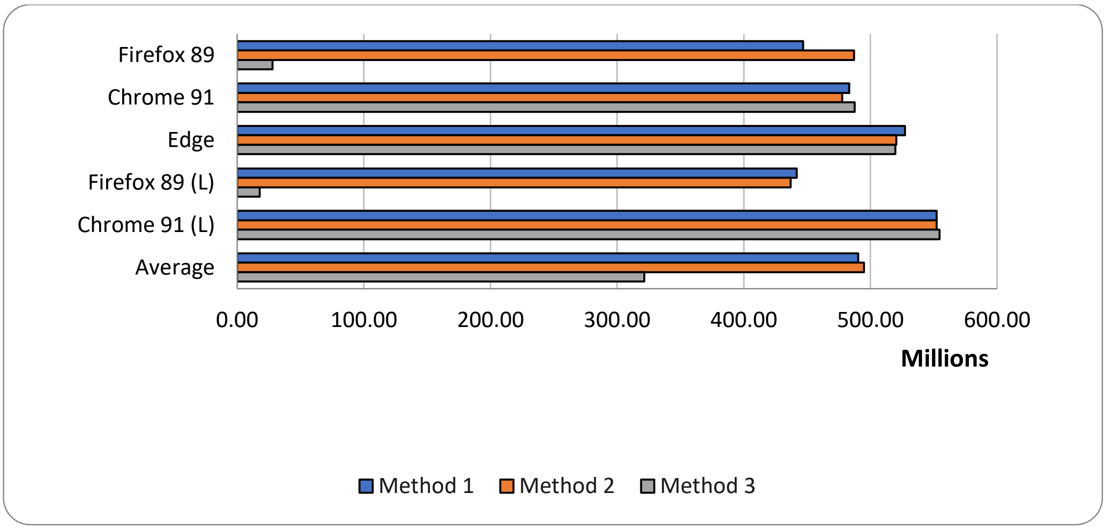

Figure 5.4: Result chart of String concatenation

5.4.2.2 Concatenate list of strings

Setup

```
let arr = [1, 2, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "abc", 92, 18, 99, 100];
```

Method 1: Using plus operator

```
let str1 = " "
for (let i=0; i<arr.length; i++) {
   str1 += arr[i]
}
```

Method 2: Using template literals

```
let str1 = " "
for(let i=0; i<arr.length; i++){
   str1 += `${arr[i]}`
}
```

Method 3: Using concat function

```
let str1 = " "
for(let i=0; i<arr.length; i++){
   str1 = str1.concat(arr[i])
}
```

Method 4: Using join function

```
arr.join('')
```

	Method 1	Method 2	Method 3	Method 4
Firefox 89	112,846.47	356,272.85	248,936.13	186,400.76
Chrome 91	285,311.00	322,402.58	230,117.15	157,461.82
Edge	291,811.27	318,026.25	230,634.27	163,598.77
Firefox 89 (L)	85,678.45	206,404.76	177,299.91	148,590.06
Chrome 91 (L)	231,143.90	251,007.06	189,904.96	160,952.41
Average	201,358.22	290,822.70	215,378.49	163,400.76
Rank	3	1	2	4

Table 5.6: Results set, usage of Concatenate list of strings

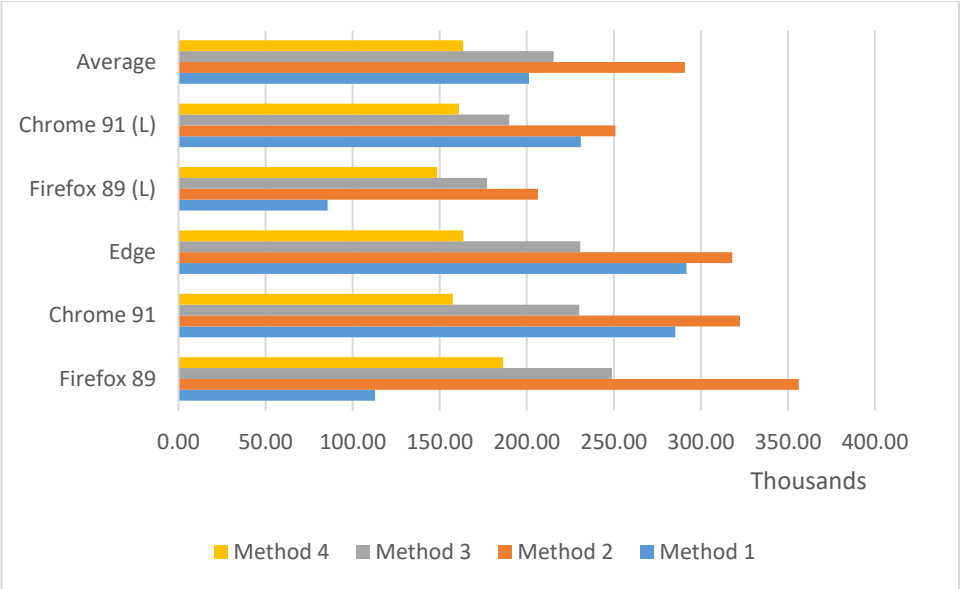

Figure 5.5: Results chart of Concatenate list of strings

5.4.2.3 Find the existence of a given term

Setup

```
const text = 'The quick brown fox jumps over the lazy dog. If the dog barked, was it really lazy?'
let found = false
const searchTerm = /\./g
```

Method 1: Using search function

```
found = text.search(searchTerm) > -1
```

Method 2: Using match function

```
found = text.match(searchTerm) !== null
```

	Method 1	Method 2
Firefox 89	20,420.14	19,188.47
Chrome 91	41,586.54	41,288.60
Edge	40,419.51	39,066.47
Firefox 89 (L)	16,181.91	15,733.37
Chrome 91 (L)	42,009.55	39,673.63
Average	32,123.53	30,990.11
Rank	1	2

Table 5.7: Results set, usage of match and search

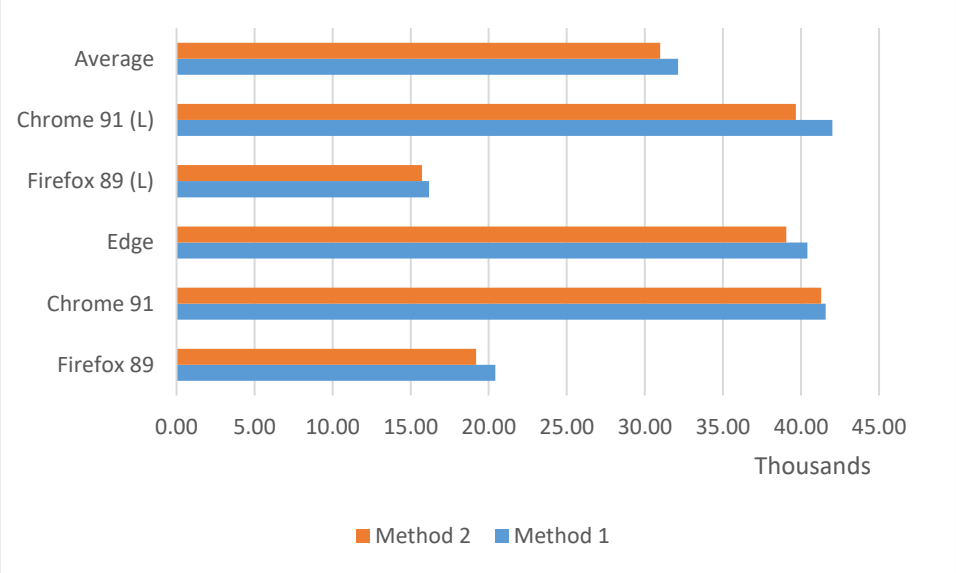

Figure 5.6: Results chart of match and search

5.4.2.4 Find a given term

Setup

```
const text = 'The quick brown fox jumps over the lazy dog. If the dog b
arked, was it really lazy?'
let found = false
const searchTerm = "lazy"
```

Method 1: Using indexOf function

```
found = text.indexOf(searchTerm) > -1
```

Method 2: Using includes function

```
found = text.includes(searchTerm)
```

Method 3: Using search function

```
found = text.search(searchTerm) > -1
```

Method 4: Using match function

```
found = text.match(searchTerm) !== null
```

	Method 1	Method 2	Method 3	Method 4
Firefox 89	16,067,835.99	18,437,444.74	8,865,444.72	4,834,623.64
Chrome 91	479,622,807.93	13,498,821.82	2,074,530.13	1,694,899.06
Edge	521,096,309.81	14,303,272.41	2,235,894.55	1,880,055.51
Firefox 89 (L)	13,131,187.20	15,475,230.21	8,470,023.54	3,251,816.54
Chrome 91 (L)	394,697,498.49	12,987,956.94	1,769,986.43	1,397,470.16
Average	284,923,127.88	14,940,545.22	4,683,175.87	2,611,772.98
Rank	1	2	3	4

Table 5.8: Results set, usage of index of, include, match and search

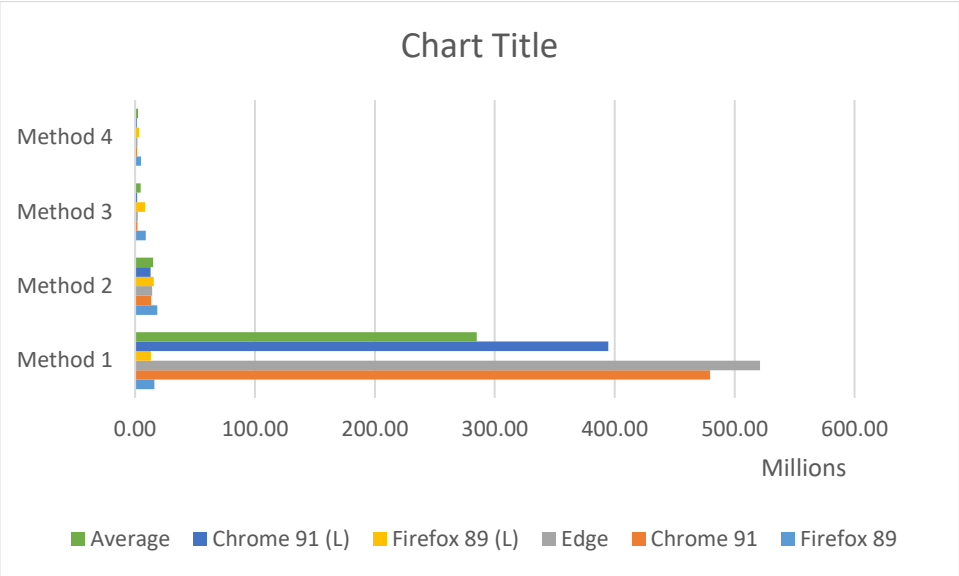

Figure 5.7: Results chart of index of, include, match and search

5.4.2.5 Splitting a string

Setup

```
let str = "NDwtAeUVUGakEtnTEobI"
let splitter = "k"
```

Method 1: Using split function

```
const splittedString = str.split(splitter)
let a = splittedString[0]
let b = splittedString[1]
```

Method 2: Using for loop and substring function

```
for(let i=0; i<str.length; i++) {
   if(str[i] === splitter) {
      let a = str.substring(0, i)
      let b = str.substring(i, str.length-1)
   }
}
```

Method 3: Using indexOf and substring functions

```
let index = str.indexOf(splitter)
let a = str.substring(0, index)
let b = str.substring(index, str.length-1)
```

Method 4: using search and substring functions

```
let index = str.search(splitter)
let a = str.substring(0, index)
let b = str.substring(index, str.length-1)
```

Method 5: Using for loop with slice function

```
for(let i=0; i<str.length; i++) {
   if(str[i] === splitter) {
      let a = str.slice(0, i)
      let b = str.slice(i, str.length-1)
   }
}
```

Method 6 : Using indexOf and slice functions

```
let index = str.indexOf(splitter)
let a = str.slice(0, index)
let b = str.slice(index, str.length-1)
```

Method 7: Using search and slice functions

```
let index = str.search(splitter)
let a = str.slice(0, index)
let b = str.slice(index, str.length-1)
```

	Method 1	Method 2	Method 3	Method 4
Firefox 89	3,610,713.23	3,487,576.04	8,862,557.41	6,932,318.43
Chrome 91	3,829,588.26	3,091,028.20	526,113,039.16	2,325,459.49
Edge	3,521,395.95	2,638,669.92	485,818,830.26	2,141,701.90
Firefox 89 (L)	2,351,234.25	3,548,268.07	6,603,045.99	5,340,876.57
Chrome 91 (L)	3,067,613.34	2,142,773.78	395,474,058.02	1,828,728.12
Average	3,276,109.00	2,981,663.20	284,574,306.17	3,713,816.90
Rank	5	7	1	3

	Method 5	Method 6	Method 7
Firefox 89	4,244,495.11	8,782,214.68	6,830,910.76
Chrome 91	2,834,157.11	289,865,965.05	2,441,265.96
Edge	2,752,755.69	263,856,761.55	2,108,726.66
Firefox 89 (L)	3,374,000.13	6,556,198.71	5,266,734.05
Chrome 91 (L)	2,154,000.44	214,915,301.40	1,787,941.56
Average	3,071,881.70	156,795,288.28	3,687,115.80
Rank	6	2	4

Table 5.9: Results set, Splitting a string

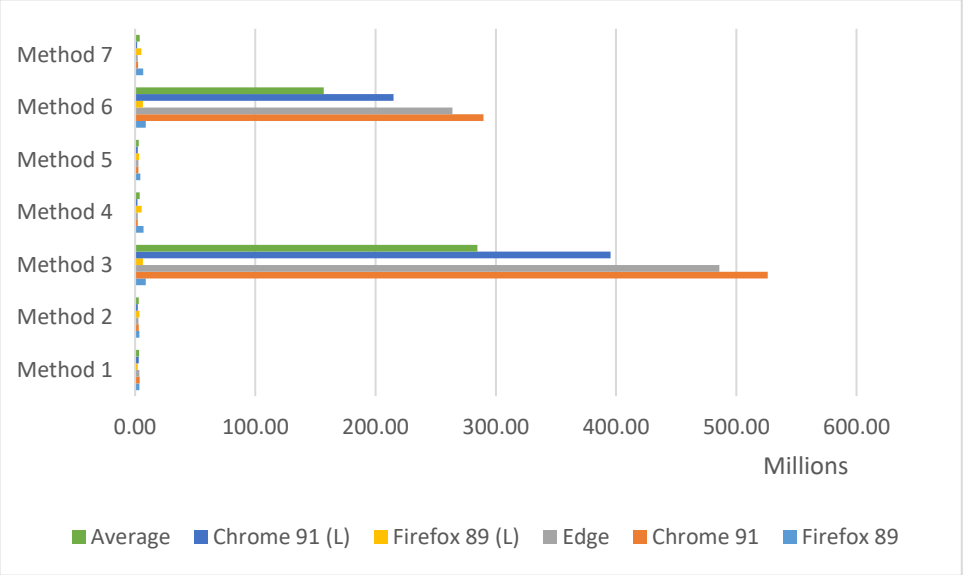

Figure 5.8: Result chart of Splitting a string

5.4.3 Array Operations

Arrays play a vital role since it is the primary data structure of JavaScript. Array is the most common candidate in general programming to hold collection of data in JavaScript. Therefore, the performance of arrays has an enormous effect on the

performance of a JavaScript application.

5.4.3.1 Search an element of an array

Setup

```
const textArr = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy", "dog
", ". ", "If", "the", "dog", "barked,", "was", "it", "really", "lazy?"]
const searchTerm = "dog"
let found = false
```

Method 1: using a for loop

```
for(let i=0; i<textArr.length; i++){
   if(searchTerm === textArr[i]) {
      found = true
      break
   }
}
```

Method 2: Using find function

```
found = textArr.find(element => element === searchTerm) !== undefined
```

Method 3: Using indexOf function

```
found = textArr.indexOf(searchTerm) > -1
```

Method 4: Using findIndex function

```
found = textArr.findIndex(element => element === searchTerm) > -1
```

Method 5: includes function

```
found = textArr.includes(searchTerm)
```

Method 6: Using some function

```
found = textArr.some(element => element === searchTerm)
```

_	Method 1	Method 2	Method 3	Method 4
Firefox 89	22,056,457.13	4,345,391.71	8,729,148.08	4,327,049.73
Chrome 91	39,996,547.04	23,262,723.14	33,323,128.25	23,765,651.19
Edge	36,497,714.39	22,076,212.63	30,840,630.27	21,731,527.31
Firefox 89 (L)	21,341,255.08	3,851,077.26	12,498,679.00	3,950,353.45
Chrome 91 (L)	26,728,675.63	14,666,273.72	22,362,139.44	14,518,280.09
Average	29,324,129.85	13,640,335.69	21,550,745.01	13,658,572.35
Rank	2	5	3	4

_	Method 5	Method 6
Firefox 89	14,389,804.54	3,307,764.75
Chrome 91	601,503,029.58	25,012,048.25
Edge	562,912,171.45	22,547,656.07
Firefox 89 (L)	13,140,264.61	2,675,549.47
Chrome 91 (L)	412,023,822.94	14,264,813.53
Average	320,793,818.62	13,561,566.41
Rank	1	6

Table 5.10: Results set, Search an element of an array

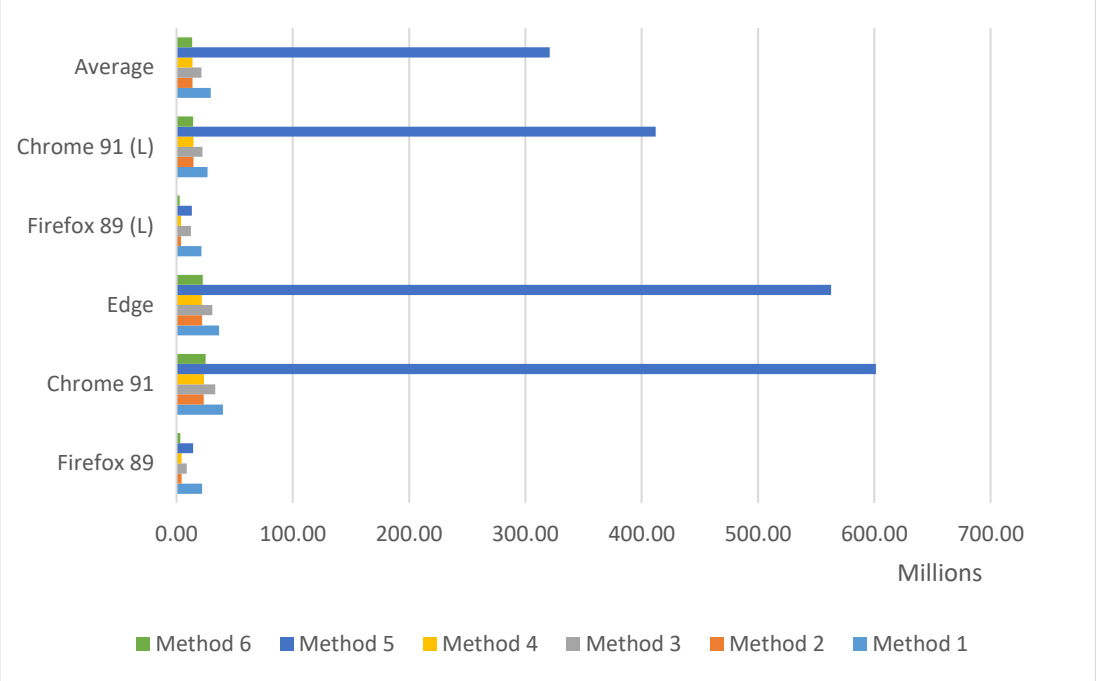

Figure 5.9: Results chart of Search an element of an array

5.4.3.2 Search an object element of an array

Setup

Setup array contained 100 objects with random values. To shorten the text, only 3 objects are shown here.

```
const objectArr = [
   {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435
   },
   {
    "prop1": "zwzskp",
    "prop2": 0.9607628198461381
   },
   {
    "prop1": "v2vnh",
    "prop2": 0.1838385705385971
   },...
 ]
 const searchTerm = "a8i98"
 let found = false
```

Method 1: Using a for loop

```
for(let i=0; i<objectArr.length; i++){
   if(searchTerm === objectArr[i].prop1) {
      found = true
      console.log(found)
      break
   }
}
```

Method 2: Using find function

```
found = objectArr.find(element => element.prop1 === searchTerm) !== undefined
console.log(found)
```

Method 3: Using findIndex function

```
found = objectArr.findIndex(element => element.prop1 === searchTerm) > -1
console.log(found)
```

Method 4: Using some function

```
found = objectArr.some(element => element.prop1 === searchTerm)
console.log(found)
```

	Method 1	Method 2	Method 3	Method 4
Firefox 89	27,067.24	26,982.23	26,853.78	24,658.27
Chrome 91	35,001.67	33,816.45	33,583.04	34,085.06
Edge	32,458.32	31,183.41	30,635.21	31,195.91
Firefox 89 (L)	15,256.77	16,222.82	15,927.94	14,637.12
Chrome 91 (L)	30,640.36	27,577.98	26,956.23	27,821.12
Average	28,084.87	27,156.58	26,791.24	26,479.50
Rank	1	2	3	4

Table 5.11: Results set, Search an object element of an array

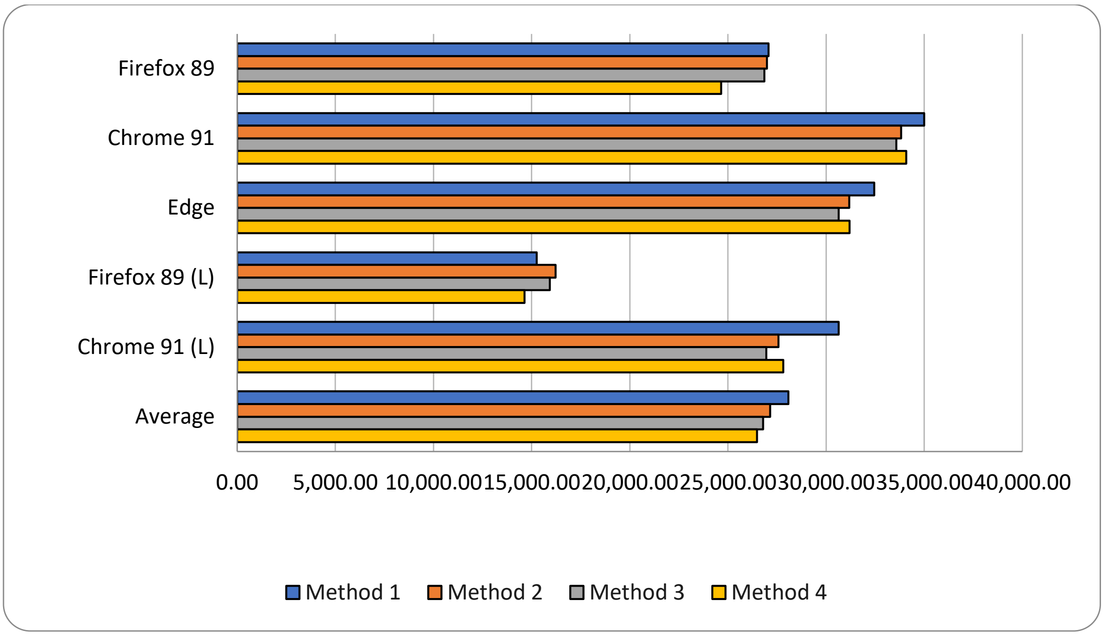

Figure 5.10: Results chart of Search an object element of an array

5.4.3.3 Adding an element to array by index

Setup

```
let arr = [1, 2, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 99, 100];

const index = 42
const newValue = "abc"
```

Method 1: Using slice function

```
let firstArr = arr.slice(0,index)
firstArr.push(newValue)
firstArr.concat(arr.slice(index))
```

Method 2: using splice function

```
arr.splice(index, 0, newValue)
```

	Method 1	Method 2
Firefox 89	332,148.08	684,241.38
Chrome 91	668,845.23	866,837.25
Edge	714,063.37	894,840.03
Firefox 89 (L)	192,707.96	471,710.77
Chrome 91 (L)	332,536.74	460,732.04
Average	448,060.28	675,672.29
Rank	2	1

Table 5.12: Results set, adding an element to array by index

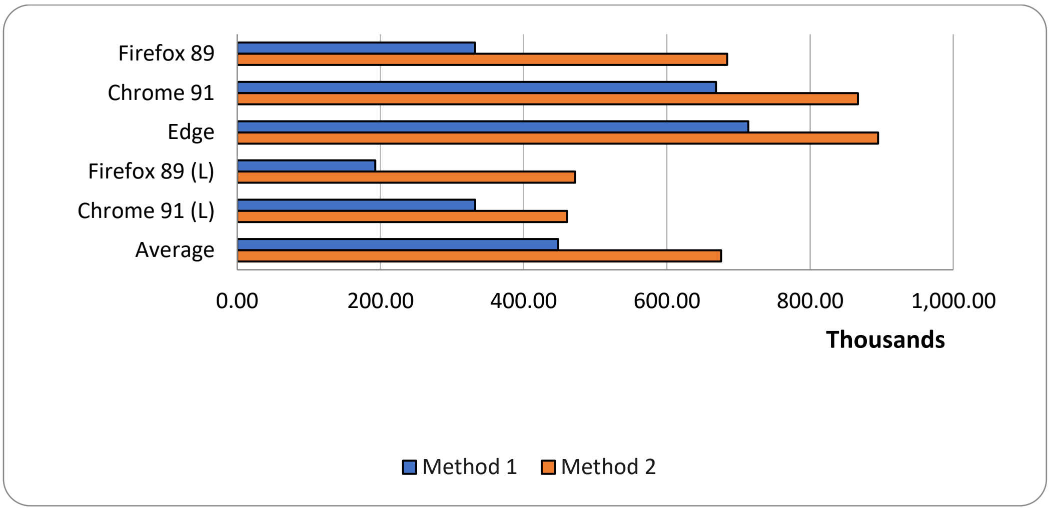

Figure 5.11: Results chart of adding an element to array by index

5.4.3.4 Remove an element from an array

Setup

```
let arr = [1, 2, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, "hu", 92, 18, 99, 100];
const index = 42
```

Method 1: Using slice function

```
const res = (arr.slice(0,index)).concat(arr.slice(index+1))
```

Method 2: Using splice function

```
arr.splice(index, 1)
```

	Method 1	Method 2
Firefox 89	291,802.87	645,800.72
Chrome 91	816,508.19	1,815,857.14
Edge	805,186.05	1,807,054.48
Firefox 89 (L)	186,592.97	449,182.27
Chrome 91 (L)	551,519.58	1,069,676.54
Average	530,321.93	1,157,514.23
Rank	2	1

Table 5.13: Results set, remove an element from an array

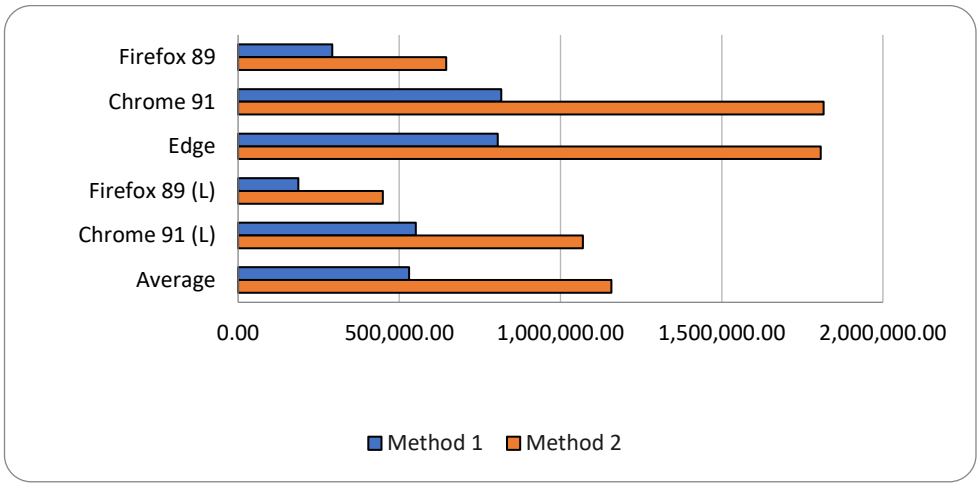

Figure 5.12: Results chart of remove an element from an array

5.4.3.5 Concatenate Arrays

Setup

Setup contains another array of same size with random values which is not given here in order to shorten the text.

```
let arr1 = [1, 2, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, " abc", 92, 1
8, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, " abc", 92, 18, 4 , 5, 66, 3
8, 39, 3993, 33, "test", "hello", 93, 93, 20, 77, " abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 3
3, "test", "hello", 93, 93, 20, 77, " abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hel
lo", 93, 93, 20, 77, " abc", 92, 18, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello", 93, 93, 2
0, 77, " abc", 92, 18, 99, 100];
```

Method 1: Using concat function

```
const arr3 = arr1.concat(arr2)
console.log(arr3.length)
```

Method 2: Using flat function

```
arr1.push(arr2)
const arr3 = arr1.flat()
console.log(arr3.length)
```

Method 3: Using a for loop

```
for(let i=0; i<arr2.length; i++) {
   arr1.push(arr2[i])
}
console.log(arr1.length)
```

	Method 1	Method 2	Method 3
Firefox 89	20,919.03	19,783.97	20,674.43
Chrome 91	35,004.12	6,747.91	27,395.18
Edge	42,048.51	9,361.24	28,494.88
Firefox 89 (L)	11,371.57	10,912.12	11,157.89
Chrome 91 (L)	33,219.16	7,947.46	28,163.87
Average	28,512.48	10,950.54	23,177.25
Rank	1	3	2

Table 5.14: Results set, concatenate arrays

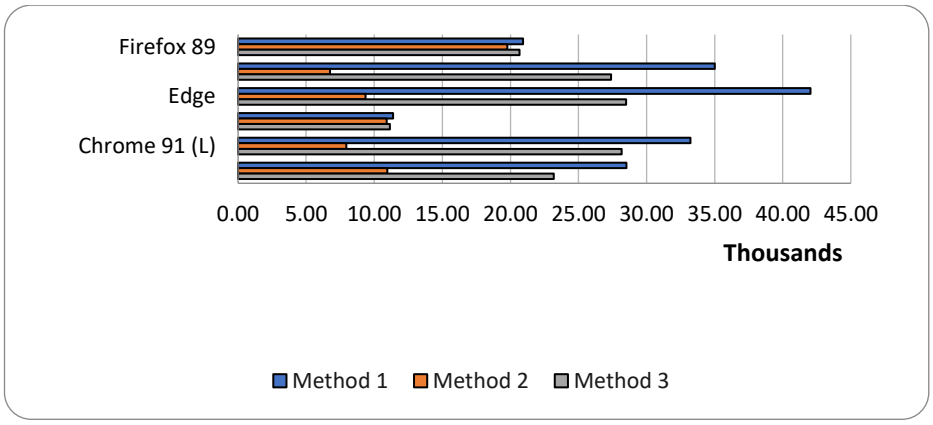

Figure 5.13: Results chart of concatenate arrays

5.4.4 Object Operations

5.4.4.1 Find an Add/search/edit/delete a prop of object in a list by index

Setup

Setup array contained 100 objects with random values. To shorten the text, only 3 objects are shown here.

```
const objectArr = [
   {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435
   },
   {
    "prop1": "zwzskp",
    "prop2": 0.9607628198461381
   },
   {
    "prop1": "v2vnh",
    "prop2": 0.1838385705385971
   },...
 ]
const givenIndex = 50
```

Method1: Using for..of

```
let index = -1;
for (let val of arr) {
   ++index;
   if(index === givenIndex) {
      value = val.prop2;
   }
}
console.log(value)
```

Method2: Using for..in

```
for (let index in arr) {
  if(index === givenIndex) {
     value = arr[index].prop2;
  }
}
console.log(value)
```

	Method 1	Method 2
Firefox 89	25,352.40	18,474.60
Chrome 91	33,651.44	22,687.75
Edge	32,989.42	24,173.35
Firefox 89 (L)	11,754.74	8,559.27
Chrome 91 (L)	33,071.38	25,415.99
Average	27,363.88	19,862.19
Rank	1	2

Table 5.15: Result set, find an add/search/edit/delete a prop of object in a list by index

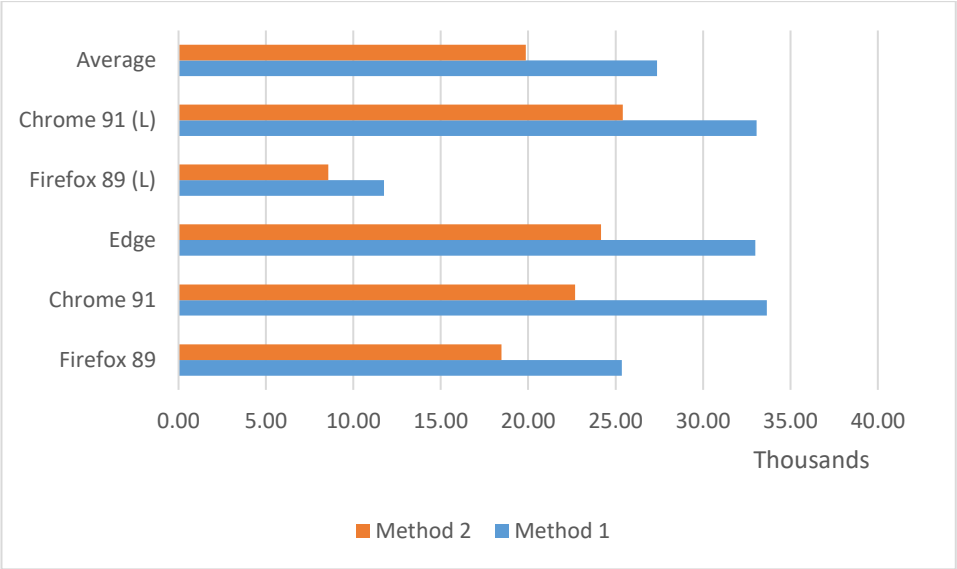

Figure 5.14: Results chart of find an add/search/edit/delete a prop of object in a list by index

5.4.4.2 Merge Objects

Setup

```
const obj1 = {
  "prop1": "n80bc",
  "prop2": 0.046610195214435435,
  "prop3": 12345
}

const obj2 = {
  "prop4": "abcd",
  "prop5": 126.436
}
```

Method1: Using object destructuring

```
const obj3 = {...obj1, ...obj2}
```

Method2: Using object assign function

```
const obj3 = Object.assign({}, obj1, obj2)
```

Method3: Using destructring and for loop

```
const obj3 = {...obj1}
let keys = Object.keys(obj2)
for (let i=0; i<keys.length; i++) {
   obj3[keys[i]] = obj2[keys[i]]
}
```

Method4: Using assign function and for loop

```
const obj3 = Object.assign({}, obj1)
let keys = Object.keys(obj2)
for (let i=0; i<keys.length; i++) {
   obj3[keys[i]] = obj2[keys[i]]
}
```

_	Method 1	Method 2	Method 3	Method 4
Firefox 89	1,893,553.14	3,576,076.96	2,248,308.29	2,530,066.96
Chrome 91	328,269.06	2,978,373.49	316,302.25	2,474,042.06
Edge	334,265.39	2,811,180.58	296,210.96	2,329,521.22
Firefox 89 (L)	1,671,669.85	2,579,661.80	1,628,885.42	1,828,234.93
Chrome 91 (L)	287,202.57	2,357,432.08	252,841.89	1,765,544.02
Average	902,992.00	2,860,544.98	948,509.76	2,185,481.83
Rank	4	1	3	2

Table 5.16: Result set, Merge Objects

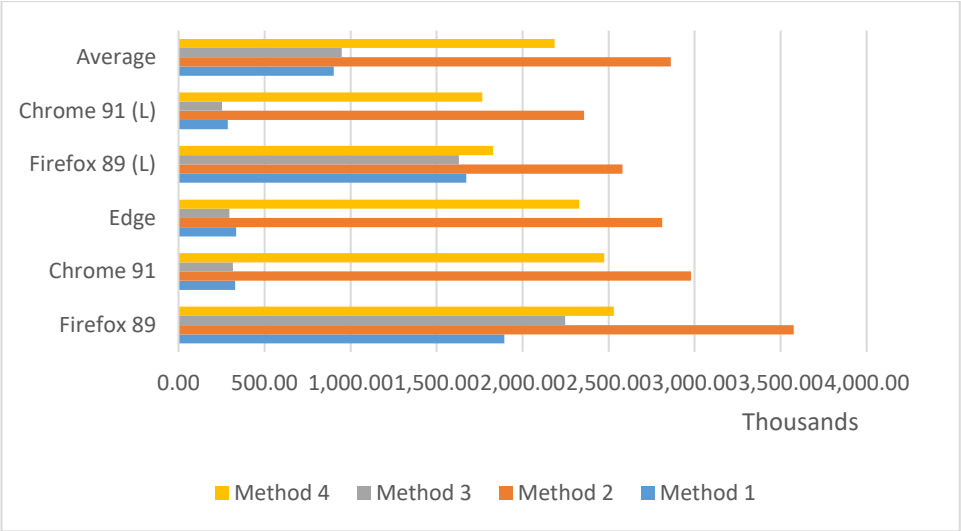

Figure 5.15: Results chart of Merge Objects

5.4.4.3 Add a Property to an Object

Setup

```
const obj = {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435
  }
```

Method1

```
obj.prop3 = 12345
```

Method2

```
obj['prop3'] = 12345
```

Method3: Using defineProperty method

```
Object.defineProperty(obj, 'prop3', {
   value: 12345
 })
```

	Method 1	Method 2	Method 3
Firefox 89	396,872,047.19	410,156,393.46	8,994,094.22
Chrome 91	144,838,123.68	143,526,220.82	3,622,995.08
Edge	510,377,897.35	510,144,251.38	3,527,485.13
Firefox 89 (L)	280,576,576.04	289,472,068.06	5,124,245.38
Chrome 91 (L)	424,825,984.40	423,492,040.37	3,141,229.31
Average	351,498,125.73	355,358,194.82	4,882,009.82
Rank	2	1	3

Table 5.17: Result set, Add a Property to an Object

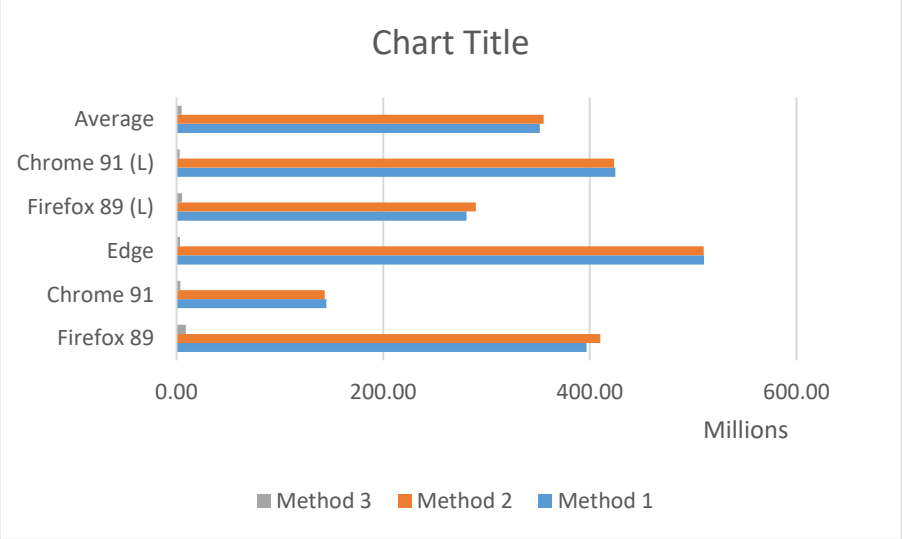

Figure 5.16: Result chart of Add a Property to an Object

5.4.4.4 Remove a property from an object

Setup

```
const obj = {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435,
    "prop3": 12345
  }
```

Method1: Using delete function

```
delete obj.prop3
```

Method2: Using object destructuring

```
const {prop3, ...newObj} = obj
```

	Method 1	Method 2
Firefox 89	19,998,277.98	3,683,946.44
Chrome 91	9,427,040.11	1,045,298.19
Edge	7,674,652.81	1,075,844.64
Firefox 89 (L)	13,035,144.62	2,433,015.06
Chrome 91 (L)	8,562,133.59	971,107.71
Average	11,739,449.82	1,841,842.41
Rank	1	2

Table 5.18: Result set, Remove a property from an object

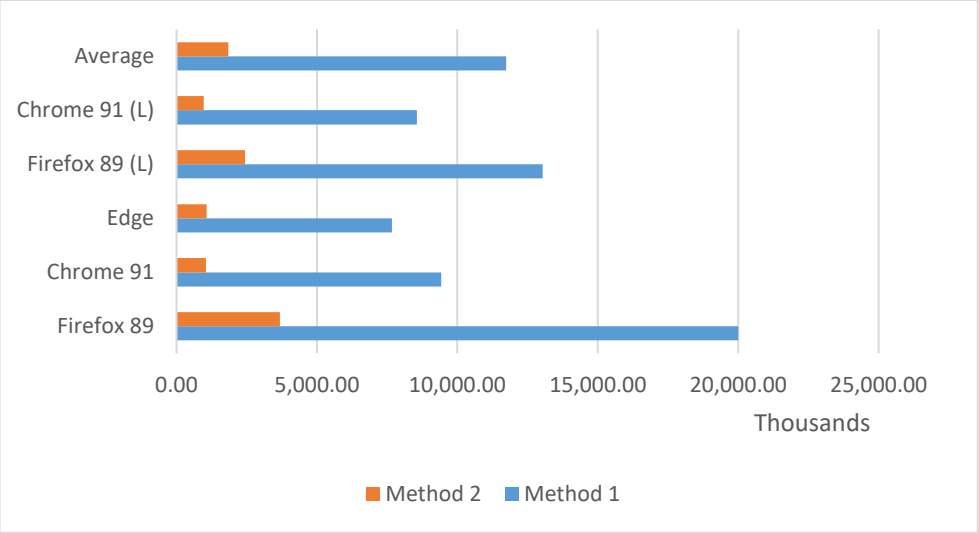

Figure 5.17: Result chart of Remove a property from an object

5.4.4.5 Check the Existence of a Prop

Setup

```
const obj = {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435,
    "prop3": 12345
  }
```

Method1: Using hasOwnProperty function

```
obj.hasOwnProperty('prop3')
```

Method2: Using in operator

```
'prop3' in obj
```

Method3: Checking if undefined

```
obj.prop3 !== undefined
```

	Method 1	Method 2	Method 3
Firefox 89	498,816,187.61	617,022,974.39	633,302,295.91
Chrome 91	57,668,932.08	538,608,615.31	540,297,562.47
Edge	54,244,736.09	508,879,881.47	517,144,033.23
Firefox 89 (L)	440,083,505.58	440,046,519.35	440,305,656.48
Chrome 91 (L)	44,107,434.91	431,446,015.22	429,277,115.95
Average	218,984,159.25	507,200,801.15	512,065,332.81
Rank	3	2	1

Table 5.19: Result set, Check the Existence of a Prop

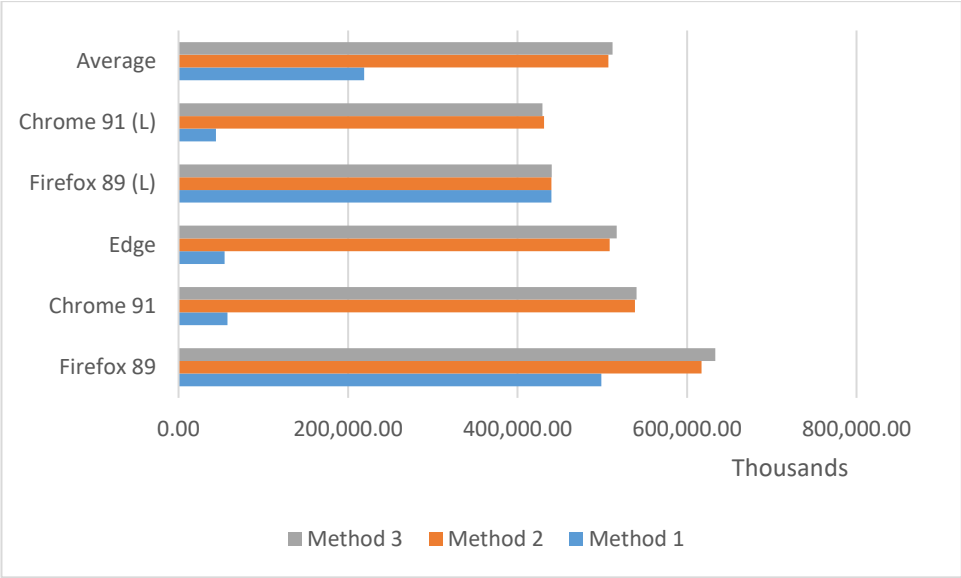

Figure 5.18: Result chart of Check the Existence of a Prop

5.4.5 Iteration

5.4.5.1 For loop

Setup

```
var arr = [1, 2, 4 , 5, 66, 38, 39, 3993, 33, "test", "hello"];
for (var i=0;i<1000000;i++) {
  arr.push(i);
}
```

Method 1: Using native JavaScript for loop

```
for (var x = 0; x < arr.length; x++) {
   value = arr[x];
}
```

Method 2: Using cached for loop

```
var l = arr.length;
for (var x = 0; x < l; x++) {
   value = arr[x];
}
```

Method 3: Using for..of statement

```
for (let val of arr) {
   value = val;
}
```

	Method 1	Method 2	Method 3
Firefox 89	446,867,226.18	487,037,074.94	27,829,354.81
Chrome 91	483,244,412.67	477,765,226.32	487,716,717.22
Edge	527,259,792.00	520,418,727.84	519,833,093.12
Firefox 89 (L)	441,850,469.92	437,118,689.71	17,836,481.49
Chrome 91 (L)	552,448,613.08	552,356,656.80	554,642,494.78
Average	490,334,102.77	494,939,275.12	321,571,628.28
Rank	2	1	3

Table 5.20: Result set, For Loop

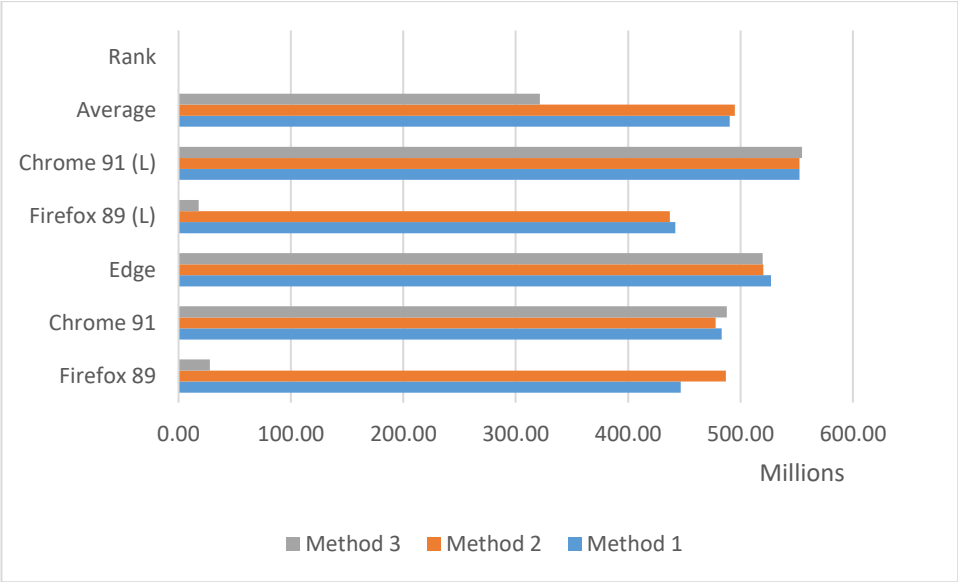

Figure 5.19: Result chart of For Loop

5.4.6 Asynchronous functions

Setup

```
function getPromise() {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            reject('foo')
        }, 500)
    })
}
```

Method1: Using async await

```
let value
  async function asyncFunc() {
     try{
        value = await getPromise()
     } catch(e) {
        value = e
     }
}
asyncFunc()
```

Method2: Using promises

```
let value
  const a = getPromise()
  a.then(
     (a)=> value=a
     )
     .catch(
        (b)=>value=b
     )
```

	Method 1	Method 2
Firefox 89	68,426.33	71,566.35
Chrome 91	112,507.21	114,473.92
Edge	110,268.99	115,975.24
Firefox 89 (L)	36,967.63	40,911.99
Chrome 91 (L)	86,845.82	96,193.55
Average	83,003.20	87,824.21
Rank	2	1

Table 5.21: Result set Asynchronous functions

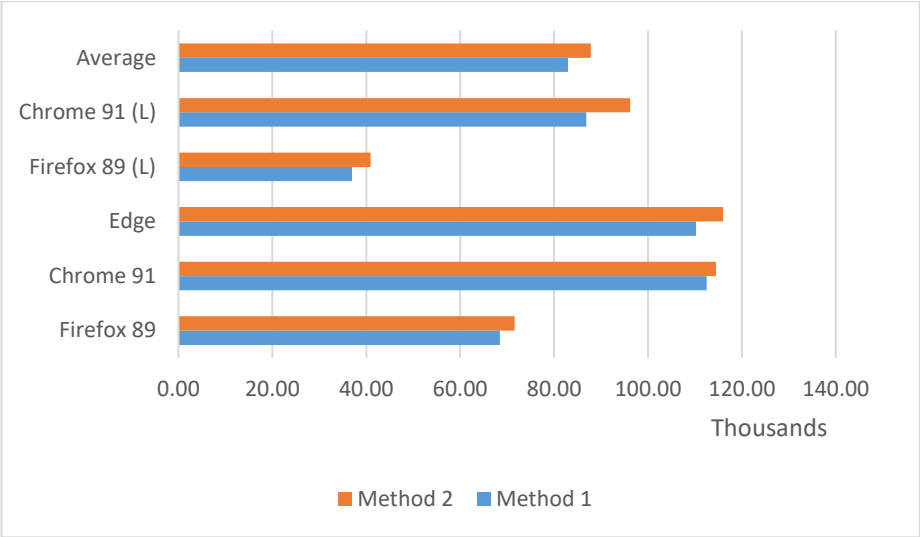

Figure 5.20: Result chart of Asynchronous functions

5.4.7 Other special operations

5.4.7.1 Equals Operation

Setup

```
const obj1 = {
    "prop1": "n80bc",
    "prop2": 0.046610195214435435,
    "prop3": 12345
  }

 const obj2 = {
    "prop4": "abcd",
    "prop5": 126.436
 }

 const a = "0.046610195214435435"
 const b = 0.046610195214435435
```

Method1: Checking without considering type

```
const out = a == b
console.log(out)
```

Method2: Checking with considering type

```
const out = a === b
console.log(out)
```

	Method 1	Method 2
Firefox 89	29,914.51	30,838.71
Chrome 91	40,692.63	45,301.90
Edge	38,919.15	43,959.44
Firefox 89 (L)	15,342.53	15,864.23
Chrome 91 (L)	41,755.06	43,716.79
Average	33,324.78	35,936.22
Rank	2	1

Table 5.22: Result set of Equals Operation

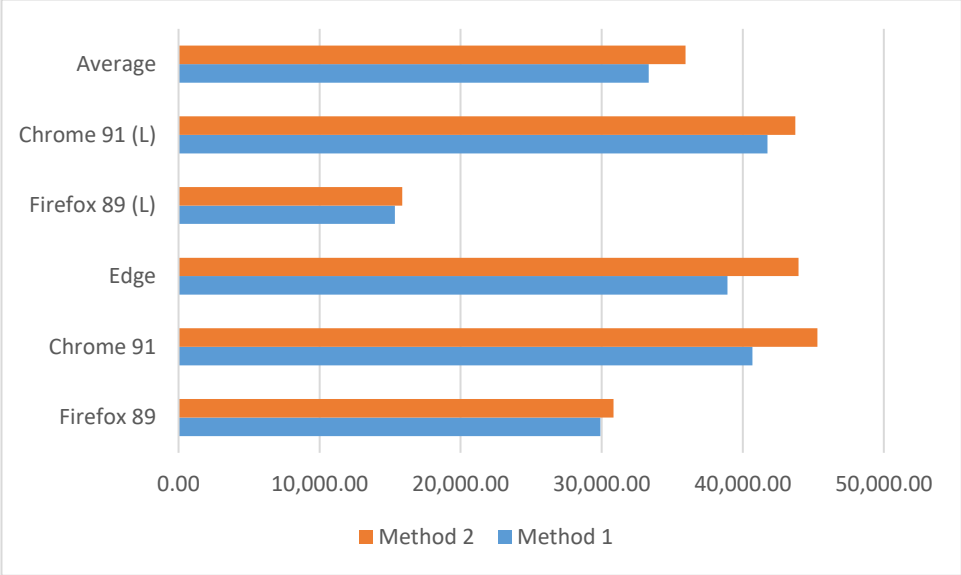

Figure 5.21: Result chart of Equals Operation

CHAPTER 6

CONCLUSION

6.1 Research Findings and Concerns

JavaScript performance improvements are getting popular as an area of research and development. There are various usages of JavaScript. Initially the JavaScript was created as a client-side language, but eventually it had become a ubiquitous language.

Nevertheless, it has been identified by this research that all these JavaScript performances are platform agnostic or in laymen's terms, have a direct relationship with the JavaScript engine that it is running on rather than the other factors such as hardware and the operating system. Therefore, the most important part for a research with similar capacity, on JavaScript performance should be focused on variations of JavaScript engines and the performance changes and optimizations.

Within the scope of this research most common and important operations of JavaScript have been identified, analyzed and documented. There may be scenarios which have not been concerned

- Sub derivations of identified main performance scenarios
- Specific corner cases which may only evolve under uncommon business requirements

Within the course of research work, it has been identified that the JavaScript updates and the JavaScript engines are being evolved very rapidly. Hence it is very important to have a document which is dynamically updated along with this new JavaScript updates as well as the new coming JavaScript engines.

Even though the initial scope of this research was to compile a paper-based document for JavaScript performance, it has been identified that the JavaScript and its performance requirements are highly dynamic. Therefore, the scope has been extended slightly in order to cater that requirement to generate a document which improves and evolves along with the new performance requirements coming up from JavaScript

updates as well as new JavaScript engines which are being released rapidly.

Initial scope of the of this research has been covered completely within this thesis and the newly identified improved outcome is also addressed. For that, a sample website has been created and plans to improve it by continuously growing as a community based open-source project.

6.2 Future work

As the improved version of outcome this standardization document should be used as the baseline for a digitally converted web application where it features the benchmark results in a readily available manner for the reference of JavaScript developers.

A prototype web site has been created to only view the data currently. And this prototype should be evolved to become the full featured dynamic JavaScript performance documentation.

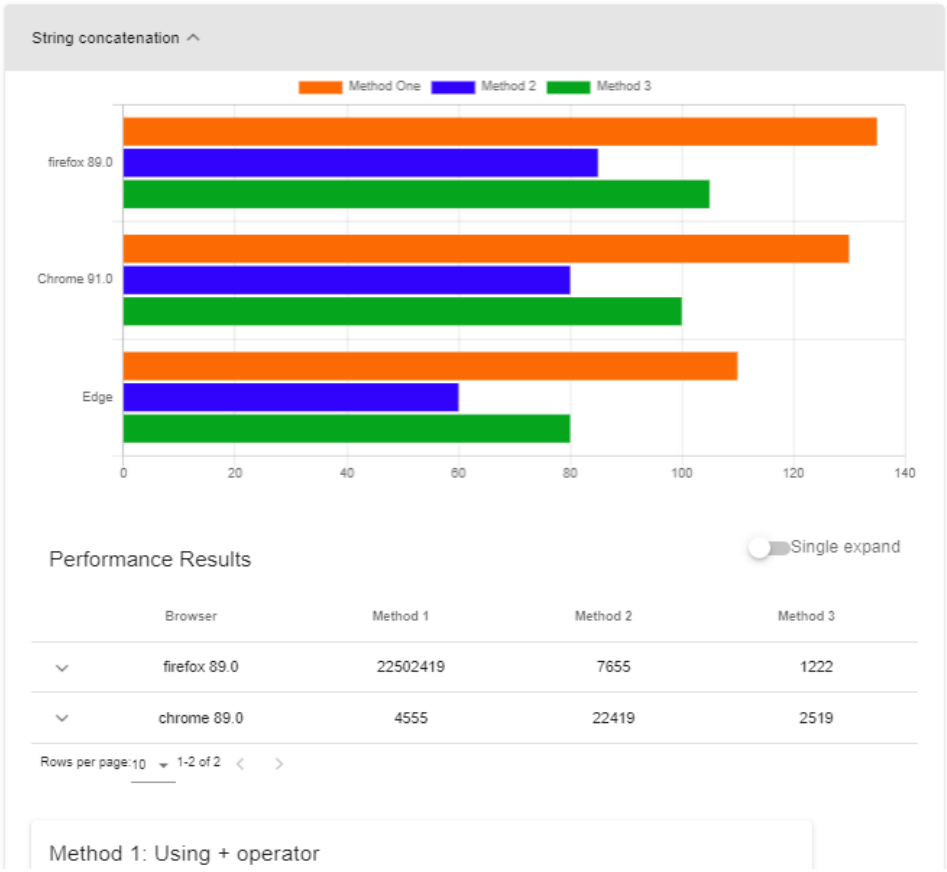

Figure 6.1 Proposed web interface of performance digital document part 1

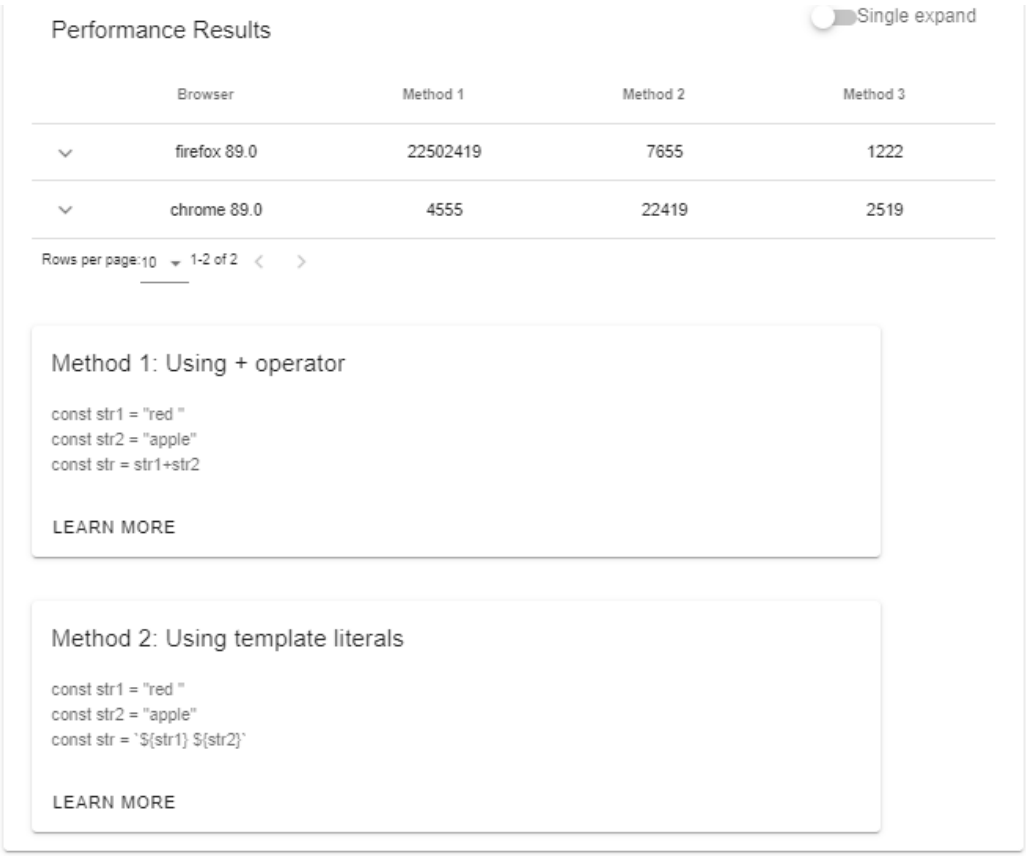

Figure 6.2: Proposed web interface of performance digital document part 2

6.2.1 Roadmap of the Web application and Future Concerns

This web application should continue to improve as a community based open-source project where each and every developer can be benefited as well as contribute at the same time to the effectiveness of the test evaluations of this performance standard documentation.

Micro benchmarks should only be allowed to add or modify by a limited set of moderators whereas other members can continuously contribute by suggesting any micro benchmark to be added or updated.

Any contributor should be able to execute the suite of benchmark on their own platform as they intended, and the web application should be able to collect the information about the platform and persist the benchmark results against the platform details for future reference.

Since JavaScript engines are highly dynamic, this documentation will also be affected by that dynamic environment. Therefore, a particular set of data may become obsolete soon. Such that it won't be much effective to generate all time average performance results. As well there may be a specific requirement to find out the performance for a given legacy JavaScript engine. Such that the best approach to calculate the average performance result is to by filtering and using a subset of data according to user specified platform details.

REFERENCES

[1] "ECMAScript." [Online]. Available: http://www.ecmascript.org/.

[2] L. Clark, "A crash course in just-in-time (JIT) compilers," Mozilla Org, 2017. [Online]. Available: https://hacks.mozilla.org/2017/02/a-crash-course-in-just-in-time-jit-compilers/.

[3] S. SOUDERS, High Performance Web sites. 2007.

[4] G. Grisogono, "JavaScript Performance Tips & Tricks," 2012. [Online]. Available: https://moduscreate.com/blog/javascript-performance-tips-tricks/.

[5] C. C. Charles D. Garrett, Jeffrey Dean, David Grove, "Measurement and Application of Dynamic Receiver Class Distributions," in Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, Washington 98195 USA, 1994.

[6] A. Gal, B. Eich, M. Shaver, and D. Anderson, "Trace-based just-in-time type specialization for dynamic languages," ACM SIGPLAN Notices, Volume 44, Issue 6, pp. 465–478, 2009.

[7] "Kraken Benchmark." [Online]. Available: http://krakenbenchmark.mozilla.org/.

[8] "Octane Benchmarks." [Online]. Available: https://developers.google.com/octane.

[9] "SunSpider Benchmarks." [Online]. Available: http://www.webkit.org/perf/sunspider/sunspider.html.

[10] W. Ahn, J. Choi, T. Shull, M. J. Garzarán, and J. Torrellas, "Improving JavaScript Performance by Deconstructing the Type System." in PLDI '14: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, Jun. 2014, pp. 496–507.

[11] Jeff Nelson, "Which language has the best future prospects: Python, Java, or JavaScript?," 2016. [Online]. Available: https://www.quora.com/Which-language-has-the-best-future-prospects-Python-Java-or-JavaScript. [Accessed: 31-Jan-2022].

[12] “V8 JavaScript Engine.” [Online]. Available: https://developers.google.com/v8/.

[13] “SpiderMonkey Project.” [Online]. Available: https://developer.mozilla.org/en- US/docs/SpiderMonkey.

[14] B. Eich, “New JavaScript Engine Module Owner.” [Online]. Available: https://brendaneich.com/2011/06/new-javascript-engine-module-owner/. [Accessed: 30-Dec-2021].

[15] “A Short History of JavaScript.” [Online]. Available: https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. [Accessed: 13-Dec-2021].

[16] “Chakra.” [Online]. Available: http://blogs.msdn.com/b/ie/archive/2021/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx.

[17] J. K. Martinsen, H. Grahn, and A. Isberg, “A Comparative Evaluation of JavaScript Execution Behavior,” in Web Engineering: 11th International Conference, ICWE 2011, Paphos, Cyprus, June 20-24, 2011, S. Auer, O. Díaz, and G. A. Papadopoulos, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 399–402.

[18] and M. K. M. S. L. Graham, P. B. Kessler, “Gprof: A call graph execution profiler,” in ACM Sigplan Notices, vol. 17, ACM, 1982, pp. 120–126.

[19] M. Fowler, “CodeSmell,” 2006. [Online]. Available: https://martinfowler.com/bliki/CodeSmell.html. [Accessed: 09-Jan-2022].

[20] R. Ashton, “you have ruined javascript,” 2014. [Online]. Available: http://codeofrob.com/entries/you-have-ruined-javascript.html. [Accessed: 09-Nov-2021].

[21] M. Selakovic and M. Pradel, “Performance issues and optimizations in JavaScript,” Proc. 38th Int. Conf. Softw. Eng. - ICSE ’16, pp. 61–72, 2016.