

**INTEGRATION OF A MEMORY ANALYZER TO THE BROWSER  
REFERENCE ARCHITECTURE**

**BY  
KAMAU HARUN KARIUKI**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE  
REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN  
COMPUTER SCIENCE**

**DEPARTMENT OF COMPUTER SCIENCE**

**MASENO UNIVERSITY**

**©2019**

## DECLARATION

This thesis is my original work and has not been submitted for a degree in any other university.

Signature: -----

Date: -----

**Harun Kariuki Kamau**

This thesis has been submitted for examination with our approval as university supervisors.

Supervised by:

Signature: -----

Date: -----

**Dr. Okoth Sylvester McOyowo**

**School of Computing and Informatics,**

**Maseno University**

Signature: -----

Date: -----

**Dr. Okoyo Henry Okora**

**School of Computing and Informatics,**

**Maseno University**

## **ACKNOWLEDGMENT**

The longest journey begins with a single step, and the peak of creativity is achieved by a team. Having come this far, I would like to first thank the Almighty Father for His grace since the time this journey begun. To the supervisors, Dr. Okoth Sylvester McOyowo and Dr. Henry Okora Okoyo, this thesis would not have been possible without you. Your guidance, dedication in reading research work and constant encouragements made it all possible. It has been an honor working with you. I acknowledge the Maseno University for giving me the opportunity to study and experience the critical knowledge in computer science. To my parents, brothers, sisters and my extended family, it is a joy being part of you and knowing that you are always praying for me.

.

## **DEDICATION**

I dedicate this research work to my family

## ABS TRACT

A Web Browser is a computer application used to access information on the World Wide Web. The browser's parsing capability has advanced over years since its inception. The advancements have consequently increased demand for memory as manifested by computer crawl. Contemporary browsers are anchored on reference architecture that lacks memory control mechanism that can limit maximum memory a browser can use thus posing a challenge in multiprogramming environments with less memory thereby making the computer to freeze. Enhanced browser reference architecture was developed for investigation. The main objective of the study was to develop and integrate a memory analyzer to the browser with a view to evaluating its performance in Web browsers. Specific objectives were to specify the functional requirements for the browser prototype, to design and develop a browser prototype, to design, implement, and integrate memory analyzer and to evaluate the performance of the memory analyzer in the developed architecture. Prototyping technique and software reuse were adopted in formulating the model. The memory analyzer component acted as a memory meter and a memory optimizer. It controlled memory hogging by limiting memory usage to a particular value set by the user and optimizing available memory by calling the garbage collector. Experiments were carried out to validate the Mozilla-based developed prototype by using Mozilla Firefox browser as a control. All tests were carried on windows environment in parallel. Memory consumption between the two browsers was recorded and statistically analyzed to test the researcher's hypothesis. To evaluate the performance of the analyzer, memory demands posed by access to popular sites such as electronic mail service providers, social networks entertainment and search engines were examined. Statistical T-test on memory consumption between the two browsers revealed that memory analyzer-integrated browser consumed 38.65 MB and 52.08 MB less with homogeneous and heterogeneous tabs respectively compared to contemporary Mozilla Firefox browser. This value is computationally significant as it provides suitable environment that facilitates concurrency in computer systems that have low memory. The study provides insights on the performance of enhanced browser reference architecture with regard to memory optimization. The study recommends further research on memory optimization approaches, as browser memory consumption is dynamic and browser technologies change often.

## TABLE OF CONTENTS

DECLARATION .....	i
ACKNOWLEDGMENT.....	ii
DEDICATION .....	iii
TABLE OF CONTENTS.....	v
LIST OF ACRONYMS .....	viii
OPERATIONAL DEFINITION OF TERMS.....	ix
LIST OF TABLES.....	x
LIST OF FIGURES .....	xi
<b>CHAPTER 1: INTRODUCTION.....</b>	<b>1</b>
1. 1 Background to the Study.....	1
1.1.1 Web Browser Evolution.....	2
1.2 Statement of the Problem.....	3
1.3 Justification of the Study .....	4
1.4 Main Objective.....	4
1.4.1 Specific Objectives .....	4
1.5 Hypothesis.....	4
1.6 Scope and Limitations.....	5
1.7 Assumptions of the Study .....	5
1.8 Significance of the study.....	5
1.9 Ethical Considerations .....	5
<b>CHAPTER 2: LITERATURE REVIEW.....</b>	<b>6</b>
2.1 Browser and Web browsing.....	6
2.2 Causes of high memory demands by web browsers. ....	6
2.2.1 Memory management in JavaScript.....	9
2.2.2 Managing memory leaks in Mozilla Firefox.....	9
2.3 Average memory consumption for web browsers .....	10
2.4 Tools to optimize the memory usage in web browsers .....	10
2.4.1 Firemin.....	11
2.4.2 Wise memory optimizer.....	12
2.4.3 SpeedyFox .....	12
2.4.4 All Browsers Memory Zip .....	13

2.5 Browser Architectures .....	14
2.5.1 Google Chrome .....	14
2.5.2 Microsoft Internet Explorer .....	16
2.5.3 Mozilla Firefox .....	17
2.6 Browser Reference Model .....	19
2.6.1 Weaknesses of the current Browser Reference Architecture .....	20
2.7 Research Gap .....	20
<b>CHAPTER 3: THE ENHANCED BROWSER REFERENCE ARCHITECTURE .....</b>	<b>22</b>
3.1 Contemporary Browser Architecture .....	22
3.1.1 User Interface .....	22
3.1.2 Browser Engine .....	22
3.1.3 Rendering Engine .....	22
3.1.4 Display/UI Backend .....	25
3.1.5 Data Persistence .....	25
3.2 The Enhanced Browser Architecture .....	26
3.2.1 Memory Analyzer .....	26
3.2.2 Flow diagram of memory analyzer .....	27
3.3 Summary .....	27
<b>CHAPTER 4: METHODOLOGY .....</b>	<b>28</b>
4.1 Research Design .....	28
4.2 Development of Browser Prototype .....	28
4.2.1 Prototyping Model .....	29
4.2.2 Development Procedure .....	29
4.2.3 Architectural Model .....	29
4.2.4 Design of the Software System .....	29
4.3 Memory Analyzer .....	34
4.3.1 Memory Analyzer Interface Design .....	34
4.3.2: Setting Memory Threshold .....	35
4.3.3: Memory Computation Logic .....	36
4.4 Experimental Setup .....	39
4.4.1 Evaluation Metrics .....	39
4.5 Population, Sample and Sampling Procedure .....	40
4.6 Data Analysis .....	40

4.7 Summary .....	40
<b>CHAPTER 5: RESULTS AND DISCUSSION.....</b>	<b>42</b>
5.1 Presentation of results .....	42
5.1.1: Memory consumption by default processes.....	42
5.1.2 Browser memory consumption .....	43
5.1.3 Memory consumption averages by MOB and Mozilla Firefox with homogeneous website tabs .....	43
5.1.4 Memory consumption by MOB and Mozilla Firefox with heterogeneous website tabs .....	51
5.1.4.2 <i>Memory consumption</i> .....	53
5.1.4.3 <i>Memory consumption</i> .....	54
5.1.4.4 <i>Variation of consumed and available memories</i> .....	55
5.2 Hypothesis Testing.....	57
5.3 Summary .....	58
<b>CHAPTER 6: CONCLUSION AND RECOMMENDATIONS .....</b>	<b>59</b>
6.1 Conclusion .....	59
6.2 Recommendations.....	60
<b>REFERENCES.....</b>	<b>61</b>
<b>APPENDICES .....</b>	<b>65</b>



## LIST OF ACRONYMS

<b>API:</b>	Application Programming Interface
<b>CERN:</b>	European Nuclear Research Center
<b>CPU</b>	Central processing Unit
<b>CSS:</b>	Cascading Style Sheet
<b>FTP:</b>	File Transfer Protocol
<b>GB:</b>	Giga Byte
<b>GC:</b>	Garbage Collector
<b>GPU:</b>	Graphical Processing Unit
<b>HTML:</b>	Hypertext Markup Language
<b>HTTP:</b>	Hypertext Transfer Protocol
<b>IE:</b>	Internet Explorer
<b>LCIE:</b>	Loosely coupled Internet Explorer
<b>MB:</b>	Megabyte
<b>MIME:</b>	Multi-Purpose Internet Mail Extensions
<b>MSDN:</b>	Microsoft Developer Network
<b>NCSA:</b>	National Center for Supercomputing Applications
<b>RAM:</b>	Random Access Memory
<b>UI:</b>	User interface
<b>URI:</b>	Uniform Resource Identifier
<b>URL:</b>	Uniform Resource Locator
<b>W3C:</b>	World Wide Web Consortium
<b>WWW:</b>	World Wide Web
<b>XML:</b>	Extensible Markup Language

## OPERATIONAL DEFINITION OF TERMS

**Virus:** It is a malicious software program or programming code that replicates by being copied or initiating its copying to another program, computer boot sector or document.

**Browser:** It is a software application for retrieving, presenting, and traversing information resources on the World Wide Web.

**Freeze:** A state that occurs when either a computer program or system ceases to respond to inputs.

**Hogging:** A state where an application takes or uses most or all part of a resource.

**Crawl:** A state where computer starts responding slowly.

**Crash:** This is an event wherein the operating system or a computer application stops functioning properly.

**Memory leak:** This is a failure in a program to release discarded memory, causing impaired performance or failure.

## LIST OF TABLES

<i>Table 5.1: Memory consumption by default processes (Research)</i> .....	42
<i>Table 5.2: Mozilla Firefox and MOB memory consumption (Research)</i> .....	43
<i>Table 5.3: MOB and Mozilla Firefox memory consumption averages with homogenous website tabs (Research)</i> .....	44
<i>Table 5.4: Browser memory consumption in MB for two tabs (Research)</i> .....	52
<i>Table 5.5: Browser memory consumption (MB) for a combination of three various websites (Research)</i> .....	53
<i>Table 5.6: Memory consumption (MB) for a combination of four or more various websites (Research)</i> .....	55
<i>Table 5.7: Variation of consumed and available memories with heterogeneous website tabs (Research)</i> .....	56

## LIST OF FIGURES

Figure 1.1: Browser timeline 1994-2010(Michal karzynski, 2010).....	3
Figure 2.1: Firemin (Brinkmann, 2014).....	11
Figure 2.2: Wise memory Optimizer(Brinkmann, 2014).....	12
Figure 2.3: SpeedyFox (Brinkmann, 2014).....	13
Figure 2.4: All browsers memory zip usage controller (Brinkmann, 2014).....	14
Figure 2.5: Google chrome architecture (Jesse et al., 2009).....	15
Figure 2.6: Internet Explorer architecture (MSDN, 2016).....	17
Figure 2.7: Mozilla Firefox architecture (Allan & Michael, 2006).....	18
Figure 2.8: Reference architecture for Web browsers (Allan & Michael, 2006).....	20
Figure 3.1: The Gecko rendering engine (Tali & Paul, 2011a).....	23
Figure 3.2: Rendering engine-Functions flow diagram (Tali & Paul, 2011b).....	24
Figure 3.3: The enhanced browser architecture (Research).....	26
Figure 3.4: The flow diagram of a memory analyzer (Research).....	27
Figure 4.1: Unpacking GeckoFx package (Research).....	31
Figure 4.2: Adding GeckoFx assembly files as references (Research).....	31
Figure 4.3: Selecting GeckoFx assembly files (Research).....	32
Figure 4.4: Adding GeckoFx browser control to the toolbox (Research).....	32
Figure 4.5: Adding GeckoFx browser control to a windows form (Research).....	33
Figure 4.6: GeckoFx browser prototype code-snippet (Research).....	33
Figure 4.7: GeckoFx browser prototype in action (Research).....	34
Figure 4.8: Memory analyzer interface design (Research).....	35
Figure 4.9: Setting memory in the memory analyzer (Research).....	36
Figure 4.10: Physical memory computation logic (Research).....	37
Figure 4.11: Memory Layout in single processor system (Research).....	38
Figure 4.12: Browser Memory consumption computation logic (Research).....	38
Figure 5.1: Variation of Available and Consumed memories with Google tabs (Research).....	46
Figure 5.2: Variation of Available and Consumed memories with YouTube tab (Research).....	48
Figure 5.3: Variation of Available and Consumed memories with Facebook tabs (Research).....	49
Figure 5.4: Variation of Available and Consumed memories with Gmail tabs (Research).....	50
Figure 5.5: Browser memory consumption for a combination of two various websites (Research).....	52
Figure 5.6: Memory consumption for a combination of three various websites (Research).....	54
Figure 5.7: Dependence of available and consumed memory on browser tabs (Research).....	57

# CHAPTER 1

## INTRODUCTION

The chapter introduces the subject under investigation. Section 1.1 gives the background of study and evolution of web browsers. Section 1.2 to 1.9 summarizes the research study under the sections aforementioned.

### **1. 1 Background to the Study**

The Internet is progressively becoming an indispensable component of today's life. Most often than not, people largely rely on the expediency and elasticity of Internet-connected devices in learning, shopping, entertainment, communication and in broad-spectrum activities, that would otherwise necessitate their physical presence (Sagar et al., 2010). To access information or services via the Internet, it requires a medium; a browser operates as a medium. It is the prime software of a computer system when the Internet is of importance. A browser retrieves, displays, and traverses information resources on the Web (World Wide Web Consortium, 2004).

Information resources comprise text, image, video, or other pieces of content. These resources are identified and accessed by a Uniform Resource Identifier (URI).

The first browser known as WorldWideWeb was made in the early 1990s by Tim Berners-Lee and later named Nexus (Tim Berners-Lee, 1999). Since then, browsers have seen tremendous advancements ranging from their architectures and usage. The earliest browsers; Nexus, Mosaic and Netscape were less complex and used considerably low computer memory (Gordon, 2017). However, they were commonly used for viewing basic HTML pages. With the advancement of the Internet, browsers have had a lot of popularity in usage globally. Today, the browser is the most used computer application in the world (Allan & Michael, 2006a; Antero et al., 2008). With limited computer power to process voluminous data generated from various sources, users have resorted to other technologies like the cloud computing and other online solutions where there is robust computer processing power, vast storage, scalability, reliability and on-demand services. In these cases, resources are accessed as services via the Internet with thin clients especially the browsers.

Originally, web information comprised a set of documents that in most cases contained text and hyperlinks to other related documents, having little or no client-side code. All rendered content originated from a single source. Web content has increasingly become more complex in pursuit to incorporate interactive features. Today, web programs have advanced to become highly

interactive applications that execute on both the server side and client machine. With these advancements, web pages today are no longer simple documents; they now comprise highly dynamic contents that work together with each other. In other words, a web page is now said to be a “system”–having dynamic contents as programs running in it, interacting with users, accessing other contents both on the web page and in the hosting browser, invoking browser Application Programming Interfaces (APIs), and interacting with programs on the server side. These advancements require adequate computer memory in order to run properly from the host computer.

Consequently, these advancements have brought along rising memory demands. In fact, memory allocation to a browser rises gradually from tens of Megabytes (MBs) to hundreds of MBs and eventually to Gigabytes (Doug, 2012). This fact only categorizes browsers as today’s memory wolfs. Indeed, it leads to browser crash. The size of RAM determines the nature of software a computing device can run and consequently the level of multiprogramming. A single process consuming nearly a gigabyte of RAM in a one GB computer will lead to starvation of other processes and therefore lower multiprogramming level and finally leads to a crawl. However, these browsers behave differently in different platforms and with the content the browser loads. Currently, web browsers have add-ons and extensions that users can use to free memory from them. This strategy does not stop the computer from freezing (Wayne, 2018).

### **1.1.1 Web Browser Evolution**

Key concepts of web browsers can be drawn back from systems envisaged by Vannevar Bush in the 1940s (Nyce & Kahn, 1991) and Ted Nelson in the 1960s (Ted Nelson, 1965). However, the World Wide Web (WWW) was first described in a proposal made by Tim Berners-Lee in 1990 at the European Nuclear Research Center (CERN), (Tim Berners-Lee, 1999). By the end of 1991, he had written the first web browser. This browser served as an HTML editor. In the same year, scholars at the University of Kansas had in parallel initiated a project on a text-only browser, which was given the name Lynx, (Legan & Dallas, 2001). Moreover, around the year, National Center for Supercomputing Applications (NCSA) developed a graphical web browser, which was branded Mosaic, (Andreessen et al, 1994). Mosaic had the capability of parsing both images and text.

As the commercial potential of the web began to grow, NCSA founded an offshoot company called Spyglass to commercialize its technologies and Mosaic's co-author left to co-found his

own company, Netscape. Later, Andreessen and his team poised to release Mosaic Netscape (Lasar and Matthew, 2011). In 1994, Berners-Lee initiated the World Wide Web Consortium (W3C), which its core mandate was to guide the development of the web and offer support on interoperability of web technologies. In the following year, Microsoft developed Internet Explorer (IE), which ignited an intense competition with Netscape. Microsoft in due course dominated the market, and Netscape released its open-source browser with the name Mozilla in 1998 (Dave Titus, 2002). Figure 1.1 illustrates a timeline of the various releases of several outstanding web browsers and their dominance in the market. Since then, Mozilla has had much advancement in its architectural design with the aim of improving its usability and security. Today, there are a number of web browsers in the market which include Galeon (Krause & Ralph, 2002), Konqueror (Nick, 2010), Maxthon (Maxthon, 2005), Avant (Avant Force, 2004), NetCaptor (Wayner & Peter, 2005), Chrome (Google, 2008), Safari (Pour & Andreas, 2003) and Opera (Opera, 2003).

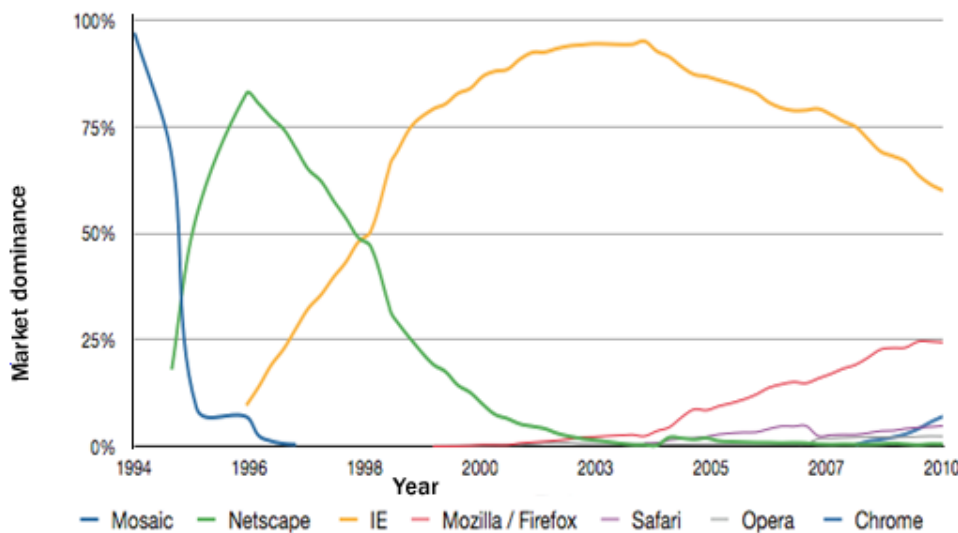


Figure 1.1: Browser timeline 1994-2010 (Michał Karzyński, 2010)

## 1.2 Statement of the Problem

Memory demand by today's browsers is overwhelming. The existing browser reference architecture does not have a control of memory usage and thus the browser continuously demands more and more memory until the operating system can no longer allocate any extra

memory making the computer to freeze. Contemporary browsers like Google Chrome, Mozilla Firefox and Internet Explorer have over years been enhanced to lower their memory consumption levels by use of extensions which end up requiring more memory. Third-party tools adopted for memory optimization by aforementioned browsers depict a similar challenge. This reduces Operating System (OS) concurrency and ultimately renders the computer unusable.

### **1.3 Justification of the Study**

Browser memory consumption is on the rise. There is a need to investigate memory demands for a browser since it has become today's application platform (Antero, 2007). The study sought to develop a memory analyzer that integrated seamlessly with the existing reference browser model with a view to evaluating its performance in Web browsers. The new model would provide a guideline to browser architects and web developers in their consideration in building intelligent applications that are highly memory optimized for sophisticated and dynamic web technologies that utilize it. In addition, web developers would adopt recommended web principles and standards for all applications they intend to develop. Consequently, the users would also change their browsing techniques and adopt methods that would improve their browsing experience.

### **1.4 Main Objective**

The main objective of the study was to develop and integrate a memory analyzer to the browser with a view to evaluating its performance in Web browsers.

#### **1.4.1 Specific Objectives**

Specific objectives of the study were to:

- i. To specify the functional requirements for the browser
- ii. To design and develop the browser prototype
- iii. To implement the design and integrate the memory analyzer
- iv. To test the performance of the memory analyzer with regard to memory optimization

### **1.5 Hypothesis**

The following hypothesis was tested at the end of the study:

*Memory consumption by analyzer-integrated browser and non-analyzer integrated browser is the same.*



## **1.6 Scope and Limitations**

The research focused on Mozilla Firefox-based browser. Global market share (Statista, 2018) places Mozilla Firefox in position two after Google Chrome. Furthermore, Mozilla project is open source and hence has no proprietary restrictions. The specified browser was investigated on Windows operating system of 32-bit architecture. While there are many browsers in existence today, the absolute scale of material to cover together with the pace at which browser technologies change lead to these limitations.

## **1.7 Assumptions of the Study**

The study was based on the following assumptions:

- i) The computer system to be used would be secured against viruses or any other vulnerability
- ii) Internet connection and power supply would be stable throughout the experiment period.
- iii) There would be negligible change on web content from where results were to be obtained.
- iv) Computer users have relevant skills for browser configuration
- v) The environment setup would be based on windows 32 bit with a maximum of 1 GB RAM

## **1.8 Significance of the study**

The study provides insights on the performance of enhanced browser reference architecture with regard to memory optimization and lays a foundation for further research on memory optimization approaches.

## **1.9 Ethical Considerations**

The researcher commits to ensuring the quality and integrity of the study by complying with research regulatory bodies. Refer to appendices.

## **CHAPTER 2**

### **LITERATURE REVIEW**

This chapter aims to provide what other scholars have written regarding memory optimization techniques in Web browsers. To accurately portray the developments in browser technologies with regard to memory optimization, an overview of the literature is presented in the following manner. Section 2.1 gives a historical perspective on browser technologies and web content parsing. Section 2.2 through 2.4 explores the causes of high memory demands by web browsers and tools that have been used to control memory hogging. Another important aspect in this study was a review on contemporary browser reference models, which is discussed under section 2.5. The chapter aimed at establishing the weaknesses of the reviewed tools and reference model to as to set stage for chapter 3

#### **2.1 Browser and Web browsing.**

Browser has evolved since its inception. In the present day, people play games, watch videos, run applications and so much more other activities that go beyond simple browsing. However, all these rich media content comes at a cost. Most modern browsers require bigger RAM to render dynamic interactive content. Mozilla Firefox and Google Chrome are the most popular Open Source web browsers dominating the market today. In due course, they have both increased their features, and consequently the amount of system resources they consume while running. Adding a few of the thousand extensions available for these browsers makes them consume hundreds of megabytes of memory and take up large amounts of disc space. Despite the developer's efforts to optimize memory usage, a majority of browser users still complain these browsers use far much more memory than they should (HAL9000, 2013).

Modern browsers are probably the most complex piece of user-orientated software on a home computer. Not only do they need to understand correctly formatted code but also badly formatted code. Moreover, not only do they need to execute arbitrarily complex software internally but also cope with deliberately malicious code while providing at least the illusion of security (Gordon, 2017). This fact has led to the continued use of browsers in everyday life.

#### **2.2 Causes of high memory demands by web browsers.**

As the browser gets used, it gradually takes more time to load during startup. In general, the speed might decrease, and browsing starts to slow down. This is a very frequent problem and

occurs partially because of fragmentation in the databases the browser uses (Kimak et al., 2014). In particular, if Mozilla Firefox is left running for a number of hours, consumed memory of well over a Gigabyte is observed even with only a few tabs open; a long running memory leak issue that plagues Firefox sometimes (Doug, 2012).

The more the tabs are opened, the more the RAM the browser will use. Each tab is designed to cache pictures, text and other active data, which keeps page data persistent while using multiple tabs. Of course, browsers like Chrome and Firefox have ways to turn this behavior off, but the user may not wish it to happen. Without caching, YouTube videos will not play in the background, and most real-time web applications will fail to work correctly (Brinkmann, 2018).

In an attempt to achieve greater stability and manage memory more effectively, most modern browsers launch new tabs as their own process. This practice has been seen in action with the use of Activity Monitor, Task Manager or a similar process monitoring application. However, this is only part of the picture. Browsers such as Chrome, Firefox and Safari often place plug-ins in a process of their own (Nield, 2018). Flash, for example, is known for poor memory management and giving it separate sandbox helps increase stability and better manages the amount of memory used. In fact, this is the same strategy used to sandbox extensions, which are often the biggest culprits behind memory leaks and poorly managed memory usage (Braga, 2011).

Similar to all software, the Windows operating system requires maintenance to get the most performance out of it. This is done by updating files, cleaning temp files, verifying file integrity, and removing stray registry entries after uninstalling programs – all of these help contribute to a smoother operating system (Karl, 2013a). Browser is no different. In Firefox, it is common to remove stray entries left behind in the “Firefox registry” and calling the “about: config” function to remove unnecessary add-ons. Failure to do so, over time, will lead to a slower browser trying to parse chunks of extraneous code.

The techniques used for measuring, clearing, and improving memory in operating systems similarly apply to browsers (Karl, 2013b). One of the reasons why browsers consume excessive memory and start slowing down is typically due to opening many tabs and having too many add-ons installed. Browser developers have devised browser add-ons that monitor browser’s memory and checking for any add-ons that are memory-heavy or memory-leaky. Examples of these include “**add-ons-memory**” which is an incredible add-on for measuring the memory demands of all add-ons installed in the browser (Williams, 2017).

A computer program for instance, the browser, may be optimized so that it executes more rapidly, capable of operating with less memory storage or other resources, or draw less power. In an application where memory space is at a premium, one might deliberately choose a slower algorithm in order to use less memory. Often, there is no "one size fits all" design which works well in all cases, so engineers make trade-offs to optimize the attributes of greatest interest. With regard to browsers, memory consumption has been the most intricate issue (Otto & Antonsson, 1991).

Memory leak is another problem. A memory leak happens when the browser for some reason does not release memory from objects, which are not needed any more. This may happen because of browser bugs, browser extensions problems and, much more rarely, browser developer mistakes in the code architecture. Leaks may occur because of browser extensions, interacting with the page. More importantly, a leak may occur because of two extensions interacting bugs. For instance, when Skype extension and the antivirus are enabled, memory leaks and when any of them is off, it does not (Ilya, 2011).

Memory leaks are caused by the following but not limited to:

- i. **Variable referencing**

In modern browsers, web developers are mandated to reclaim memory from variables not in use. Garbage-collected environments do not collect memory that is still being referenced to, and there are many ways to keep referencing memory without meaning to (e.g. create a closure to attach as an event handler and accidentally include a bunch of variables in that closure's scope). A web developer can solve these leaks completely by properly handling variable references in their code. A page reload typically frees up the memory (Sebrechts, 2012).

- ii. **Add-ons**

If add-ons are also written in a garbage-collected language (like JavaScript), then they suffer from the same issue. However, a page reload will typically not free up this memory, so it appears as if the browser is leaking memory whereas it is actually the add-on developer's fault (Mozilla Developer Network, 2019).

- iii. **Browser engine**

All modern browser engines are written in C++, which is not garbage-collected, but uses explicit memory allocation instead. If developers allocate memory and then forget to deallocate it, the

engine leaks memory. Though it is not 100% fixed, and never will be, but it is not a huge problem anymore (Pryden, 2015).

### **2.2.1 Memory management in JavaScript**

The central concept of JavaScript memory management is a concept of reachability (Ilushin & Namiot, 2015). A distinguished set of objects are assumed reachable: these are known as the roots. Typically, these include all the objects referenced from anywhere in the call stack (that is, all local variables and parameters in the functions currently being invoked), and any global variables. Objects are kept in memory while they are accessible from roots through a reference or a chain of references. There is a Garbage Collector in the browser, which cleans memory occupied by unreachable objects. However, the browser does not clean memory immediately. Most algorithms of garbage collection free memory from time to time. The browser may also postpone memory cleanup until the certain limit is occupied.

### **2.2.2 Managing memory leaks in Mozilla Firefox**

Mozilla's Servo browser engine project is designed to improve Document Object Model (DOM) memory management, with the JavaScript garbage collector to be tasked with managing native-code DOM objects. The approach would be an alternative to reference-counting for tracking pointers between low-level DOM objects, which can bring about complications like the leaking of memory objects. Two bloggers of Mozilla research blog, Josh & McAllisterin (2014) records that they established a new approach for DOM memory management, of which they use the Rust language's exciting features which includes auto-generated trait implementations, lifetime checking, and custom static analysis plug-ins.

Giving the garbage collector responsibility for managing these DOM objects requires complex interaction between Servo's Rust code and the Spider Monkey garbage collector. "Fortunately, Rust provides some good features that let us build this in a way that's fast, secure, and maintainable," Mozilla's researchers said. Mozilla collaborated with Samsung on Servo, which is intended to leverage multicore, heterogeneous architectures, and plans to productize Servo in the 2015 timeframe (Krill, 2014a).

Memory management on the DOM is a real problem that has needed to be solved (Krill, 2014b). Mozilla's researchers say it is "an open question" of how the garbage-collected DOM will perform compared to a traditional, reference-counted DOM. "The Blink team has performed

similar experiments, but they don't have Servo's luxury of starting from a clean slate and using a cutting-edge language. We expect the biggest gains will come when we move to allocating DOM objects within the JavaScript reflectors themselves. Since the reflectors need to be traced no matter what, this will reduce the cost of managing native DOM structures to almost nothing.”

### **2.3 Average memory consumption for web browsers**

When comparing the RAM usage of today's top browsers, there are a few scenarios to take into consideration. The baseline measurement is the amount of memory used when the browser is first launched which is the ability to release RAM previously used by closed tabs and plugins back to the operating system (Braga, 2011).

A research study aimed at testing memory consumption by popular Web browsers christened the Web Browser Grand Prix, revealed some interesting data. On a Windows 7 test system, Internet Explorer 9 actually used the least amount of memory for a single tab running Google's homepage, at 24 MB, edging out Google Chrome by just 3 MB. On the upper end of that spectrum, Firefox and Safari used 60 MB and 62 MB respectively (Adam, 2011).

Investigations carried out by Rosso (2015) on Mozilla Firefox 38, Google Chrome 42, Internet Explorer 11 and Opera 29 running on Windows 8.1 had the following data for a single tab. Internet Explorer 11 took the lead in terms of its low consumption of resources, 13 MB. It was followed by Opera which consumed 78 MB. On the upper end of that spectrum, Firefox and Google Chrome used 92 MB and 195 MB respectively. With five tabs open, Google chrome remains a giant memory consumer with 310 MB. Opera come second with 179 MB. On the lower end of that spectrum, Firefox and internet Explorer used 139 MB and 99 MB respectively.

### **2.4 Tools to optimize the memory usage in web browsers**

Brinkmann (2014a) postulates that, web browsers can use a lot of memory on a computer system. Once additional webpages in tabs are opened it is noticed that memory usage gets up. Firefox does a better job at that as Chrome but both can easily consume more than 1 GB of memory. High memory usage may not be an issue if the system in use has plenty memory. If it has 4, 8, 16, or even more Gigabytes of RAM, then the computer user may never run into any memory related issues. Indeed, as many may dislike how much memory a single program is using on the system but if it is not affecting performance or other operations, there is not really anything to worry about.

Internet users, who run systems with less RAM, especially 1 Gigabyte and below, sit in a different boat. Their systems may not have enough RAM for all processes running on it, that may reduce the overall performance of the system due to caching, being used to overcome this limitation. The following desktop programs attempts to free up memory using various API calls or techniques (Brinkmann, 2014b).

### 2.4.1 Firemin

With Firemin for Firefox, browser users can effectively stop Firefox memory leaks automatically. As memory usage of this popular browser increases, the system slows down and the user are stuck with limited system resources. In fact, Firefox can use up to 500 MB of memory if a user uses the browser continuously Firemin forces Firefox to reclaim the memory allocated to it by Windows and allows the user to use Firefox in an optimized environment (Ortega, 2013).

Firemin does not do anything that Windows does not do itself when the system runs out of RAM. It calls the Windows function `EmptyWorkingSet` over and over again in a loop to free up memory. Calling the function removes as many pages as possible from the working set of the specified process. The program ships with a slider that a user can use to set the desired interval in which he/she wants it to call the function.

The SQLite database optimize function is available through the tray icon context menu and simply click “Optimize Firefox” to start the compacting process. Firefox will need to be closed to do this. This tool is compatible with Windows 2000 and above. Figure 2.1 shows Firemin in action.



Figure 2.1: *Firemin* (Brinkmann, 2014)

However, the limitations of Firemin.exe are that, the technical security rating is 30% dangerous. This is because it records keyboard and mouse inputs, monitors applications and manipulates other programs. Moreover, some malware camouflages itself as Firemin.exe, particularly when located in the C:\windows or C:\windows\System32 folder. Moreover, Firemin is only compatible with Mozilla Firefox.

### 2.4.2 Wise memory optimizer

Wise Memory Optimizer helps a user to free up and tune up the physical memory taken up by some unknown non-beneficial applications to enhance PC performance. A user can enable automatic optimization mode when the free PC memory goes below a value that he/she may specify, and make Wise Memory Optimizer run even when the CPU is idle, as well as adjust the amount of memory he/she wants to free up. Then it will optimize PC memory automatically in the background. Figure 2.2 shows wise memory optimizer in action.



Figure 2.2: Wise memory optimizer (Brinkmann, 2014)

However, this tool does not prevent the browser from hogging memory it only reclaims memory from unknown non-beneficial applications.

### 2.4.3 SpeedyFox

SpeedyFox is a tool designed specifically for compacting the SQLite database files, which will in turn reduce the time taken to read from and write to them. In addition to Firefox, which it was originally designed for, SpeedyFox, can now also compact the databases for the Chrome, Epic



Browser, SRWare Iron and Pale Moon browsers. It also supports the Mozilla Thunderbird and Skype tools as well (Serea, 2019).

Upon running the portable executable, SpeedyFox automatically detects and loads the default profile for each of the supported applications. As they are very popular these days, it is also possible to load custom profiles for Firefox or Chrome portable versions. Click the SpeedyFox menu bar and select “Add custom profile” or drag the profile folder and drop it onto the SpeedyFox window. Simply tick the application profiles to optimize and click the Optimize! button. SpeedyFox starts compacting the SQLite databases. Figure 2.3 shows SpeedyFox in action.

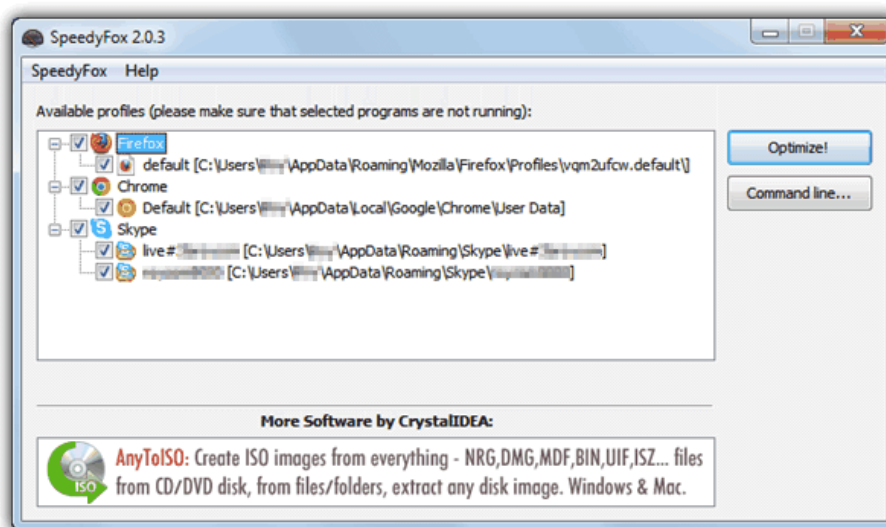


Figure 2.3: SpeedyFox (Brinkmann, 2014)

The progress window shows what databases are optimized and also how much space is saved. A user needs to make sure that the programs being optimized are not running at the time or they will not be processed. In a quick test, it reduced 14 MB of Firefox databases to 6 MB and 192 MB of Chrome databases to 186 MB. SpeedyFox developer recommends running the tool every 1-2 weeks depending on user browser usage of the included browsers.

Though tool increases Mozilla Firefox launch speed, it does not prevent memory hogging. It just clears cache over some time.

#### 2.4.4 All Browsers Memory Zip

All Browsers Memory Zip has no database compacting functions but is a dedicated memory-optimizing tool for a large number of popular web browsers. In addition to Chrome and Firefox,

it also works with other popular browsers like Opera, Internet Explorer, and Maxthon etc. The program is portable but has separate 32-bit and 64-bit versions, and when a user runs it there will be a small tooltip and then All Browsers Memory Zip will sit in the system tray optimizing the memory of any running supported browsers. Figure 2.4 illustrates All Browsers Memory Zip in action

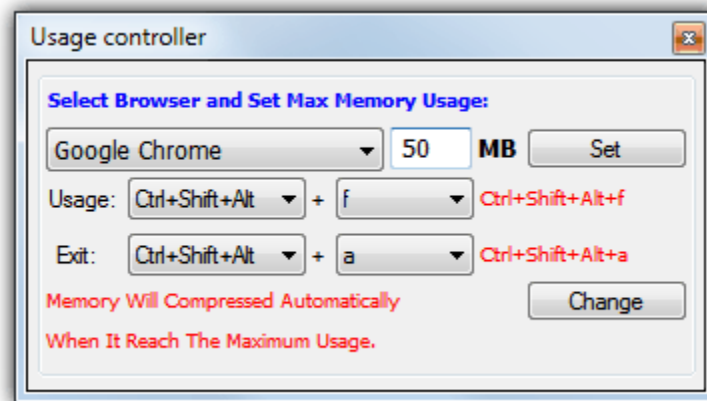


Figure 2.4: All browsers zip usage controller (Brinkmann, 2014)

Right click on the tray icon to pause the program from optimizing and pressing Usage Controller pops up the window above which enables the user to set the RAM for each browser and edit the shortcut keys. Just select the browser from the dropdown, enter the max amount in Megabytes, and click Set. This tool works in Windows XP and above.

However, this tool must execute all times a browser process is running. It requires a significant amount of memory. Consequently, it impacts negatively when streaming content over the Internet.

## 2.5 Browser Architectures

The following three browser architectures were critically explored to find out whether they are true derivations of the browser reference architecture.

### 2.5.1 Google Chrome

Google Chrome uses a multi-process architecture which gives it a competitive edge in performance over other browsers (Google, 2008). Each tab has its own process which runs independently from other tabs. Figure 2.5 illustrates the Google Chrome's major components.

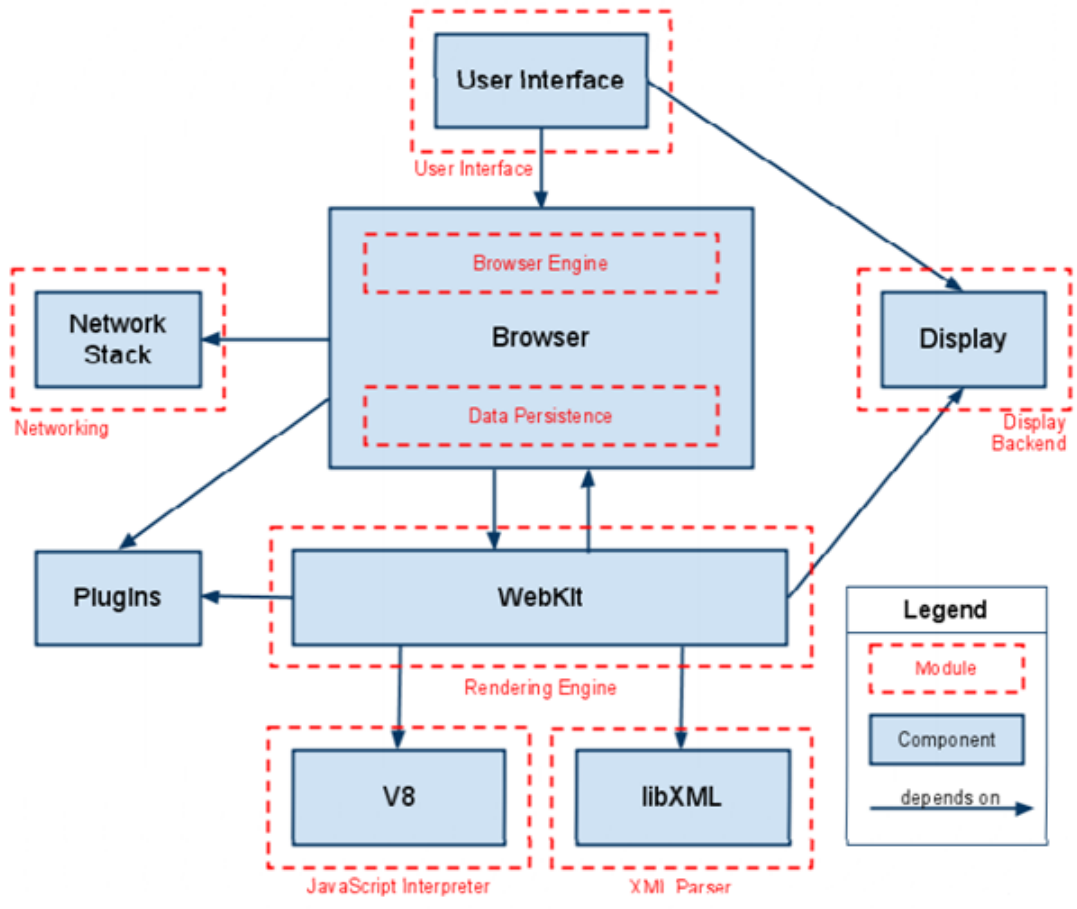


Figure 2.5: Google chrome architecture (Jesse et al, 2009)

This allows one tab process to dedicate itself to a single web-application, thereby increasing browser performance. This protects the browser application from bugs and glitches in the rendering engine. Furthermore, it restricts access from each rendering engine process to others and to the rest of the system. This scenario offers memory protection and access control as manifested in operating systems. The multi-process architecture also increases the stability of the browser, as it provides insulation. In the case that one process encounters a bug and crashes, the browser itself and the other applications running in parallel are preserved. Functionally, this is an improvement over other browsers, as highly valuable user information in other tabs will be preserved (Klein, 2019). Google Chrome has used the WebKit as a layout engine until version 27. Later versions have been using Blink. V8 has been used as JavaScript Interpreter in all versions. The components of Chrome are distributed under various open source licenses. Although Google developers have variant components in their architectural design, the browser flow logic is derived from the browser reference architecture.

## 2.5.2 Microsoft Internet Explorer

Essential to the browser's architecture is the use of the Component Object Model (COM), which governs the interaction of all of its components and enables component reuse and extensibility (MSDN, 2016). Internet Explorer uses JScript and VBScript as JavaScript interpreter and Trident layout engine. Figure 2.6 illustrates Internet Explorer's major components.

A description of each of these six components that form the architecture is as follows:

- i. **IExplore.exe** is at the top level, and is the Internet Explorer executable. It is a small application that relies on the other main components of Internet Explorer to do the work of rendering, navigation, protocol implementation, and so on.
- ii. **Browseui.dll** provides the user interface to Internet Explorer. Often referred to as the "chrome," this DLL includes the Internet Explorer address bar, status bar, menus, and so on.
- iii. **Shdocvw.dll** provides functionality such as navigation and history. It is commonly referred to as the WebBrowser control. This Dynamic-link library (DLL) exposes ActiveX Control interfaces, enabling you to easily host the DLL in a Windows application using frameworks such as Microsoft Visual Basic, Microsoft Foundation Classes (MFC), Active Template Library (ATL), or Microsoft .NET Windows Forms. When a user's application hosts the WebBrowser control, it obtains all the functionality of Internet Explorer except for the user interface provided by Browseui.dll. This means that a user needs to provide individual implementations of toolbars and menus.
- iv. **Mshtml.dll** is at the heart of Internet Explorer and takes care of its HTML and Cascading Style Sheets (CSS) parsing and rendering functionality. Mshtml.dll is sometimes referred to by its code name, "Trident". Mshtml.dll exposes interfaces that enable you to host it as an active document. Other applications such as Microsoft Word, Microsoft Excel, Microsoft Visio, and many non-Microsoft applications also expose active document interfaces so they can be hosted by shdocvw.dll. For example, when a user browses from an HTML page to a Word document, mshtml.dll is swapped out for the DLL provided by Word, which then renders that document type. Mshtml.dll may be called upon to host other components depending on the HTML document's content, such as scripting engines (for example, Microsoft JScript

or Microsoft Visual Basic Scripting Edition (VBScript)), ActiveX controls, XML data, and so on.

- v. **Urlmon.dll** offers functionality for MIME handling and code download.
- vi. **WinInet.dll** is the Windows Internet Protocol handler. It implements the HTTP and File Transfer Protocol (FTP) protocols along with cache management. Microsoft's Internet Explorer architecture utilizes the reference model components though variant in design. **IEExplorer.exe** is a wrapper for the whole application.

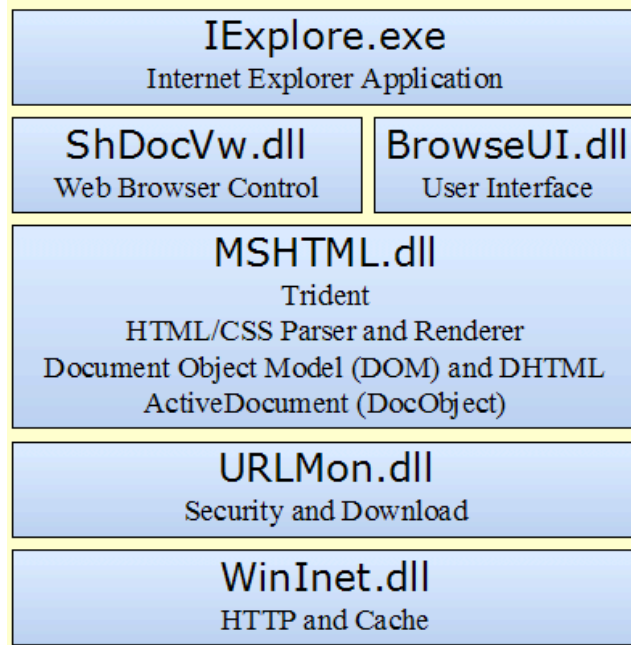


Figure 2.6: Internet Explorer architecture (MSDN, 2016)

A closer look at figure 2.6 demonstrates that Internet Explorer is derived from the contemporary browser reference architecture

### 2.5.3 Mozilla Firefox

The following model has been used in the design of Mozilla Firefox (Andre et al., 2007). Figure 2.7 illustrates Firefox major components. The User Interface is split over two subsystems; user interface and XPToolkit, allowing for parts of it to be reused in other applications in the Mozilla suite such as the mail/news client. This toolkit is a collection of loosely related facilities, from which application writers can pick and choose, which provide a platform independent API to some commonly exploited platform-specific machinery, e.g., bringing up a dialog. All data

persistence is provided by Mozilla’s profile mechanism, which stores both high-level data such as bookmarks and low-level data such as a page cache.

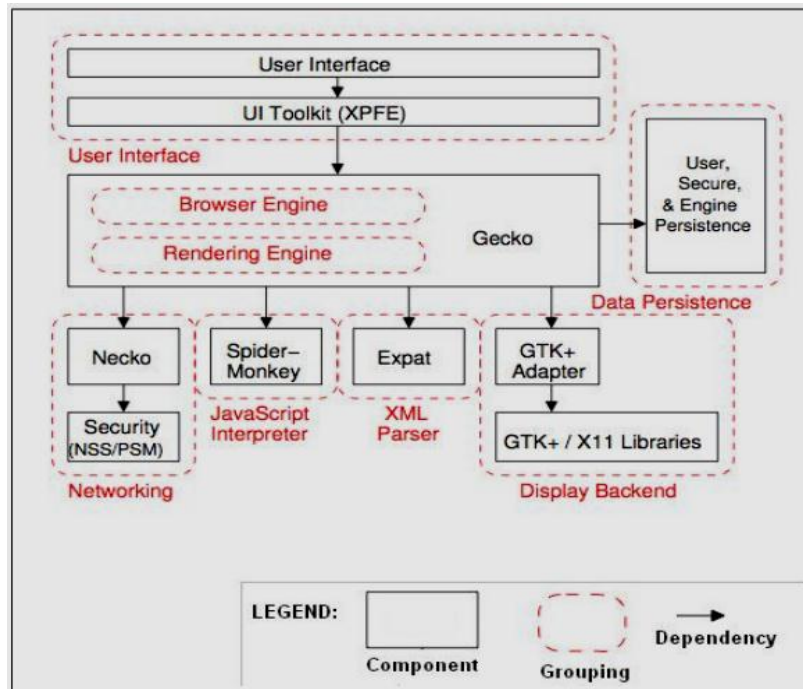


Figure 2.7: Mozilla browser architecture (Allan and Michael, 2006)

Mozilla’s Rendering Engine is larger and more complex than that of other browsers. One reason for this is Mozilla’s excellent ability to parse and render malformed or broken HTML. Another reason is that the Rendering Engine also renders the application’s cross-platform user interface. The User Interface (UI) is specified in platform-independent Extensible User Interface Language (XUL), which in turn is mapped onto platform-specific libraries using specially written adapter components. This architecture distinguishes Mozilla from other browsers in which the platform-specific display and widget libraries are used directly, and it minimizes the maintenance effort required to support multiple, diverse platforms.

In 2006, the core of Mozilla was transformed into a common runtime called XULRunner, exposing the Rendering Engine, Networking, JavaScript Interpreter, Display Backend, and Data Persistence subsystems to other applications. XULRunner allows developers to use modern web technologies to create rich client applications, as opposed to typical browser-based web applications. Mozilla developers are working on transitioning newer Mozilla-based applications such as Firefox and Thunderbird to use XULRunner directly, rather than each using a separate

copy of the core libraries (Allan & Michael, 2006). All components of this model fit exactly to those in the browser reference architecture.

## 2.6 Browser Reference Model

This study was anchored on the conceptual model in Figure 2.8, which shows the Reference architecture for web browsers (Allan & Michael, 2006). The architecture constitutes five major modules which include: User interface, Browser engine, Rendering engine, Display backend, and Data persistence. These modules work collaboratively to interpret intricate protocols and provide a visual display of the URL fetched (Siva et al., 2016).

The **User Interface** component provides the methods with which a user interacts with the Browser Engine. The User Interface provides standard features (preferences, printing, downloading, and toolbars) users expect when dealing with a desktop application.

The **Browser Engine** component provides a high-level interface to the Rendering Engine. The Browser Engine provides methods to initiate the loading of a Uniform Resource Locator (URL) and other high-level browsing actions (reload, back, forward). The Browser Engine also provides the User interface with various messages relating to error messages and loading progress.

The **Rendering Engine** component produces the visual representation of a given URL. The Rendering Engine interprets the HTML, Extensible Markup Language (XML), and JavaScript that comprises a given URL and generates the layout that is displayed in the User Interface. A prime component of the Rendering Engine is the HTML parser, this HTML parser is quite complex because it allows the Rendering Engine to display poorly formed HTML pages.

The **Networking component** provides functionality to handle URLs retrieval using the common Internet protocols of Hypertext Transfer Protocol (HTTP) and File Transfer Protocol (FTP). The Networking components handle all aspects of Internet communication and security, character set translations and MIME type resolution. The Network component may implement a cache of retrieved documents to minimize network traffic.

The **JavaScript Interpreter** component executes the JavaScript code that is embedded in a website. Results of the execution are passed to the Rendering Engine for display. The Rendering Engine may disable various actions based on user defined properties.

The **XML Parser** component is used to parse XML documents.

The **Display Backend** component is tightly coupled with the host operating system. It provides primitive drawing and windowing methods that are host operating system dependent.

The **Data Persistence** component manages user data such as bookmarks and preferences.

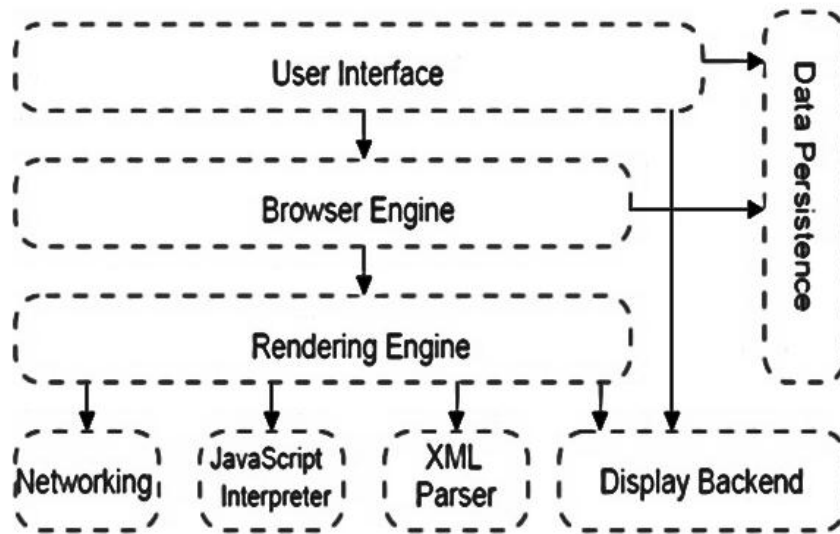


Figure 2.8: Reference architecture for web browsers (Alan and Michael, 2006)

### 2.6.1 Weaknesses of the current Browser Reference Architecture

- a) The rendering engine processes the requests made by the browser engine by rendering the fetched content provided there is little memory available for use by the browser. If the operating system can no longer allocate any more memory, the computer freezes hence becomes unusable.
- b) The browser process prevents other legitimate processes from being loaded in the main memory if it consumes almost all-available memory. This reduces the level of multiprogramming.

### 2.7 Research Gap

Based on the review of the above literature, it is evident enough that memory hogging among various browsers remains a thorny issue. In attempt to reduce its impact on computers, third party applications have been developed in quest to reduce memory consumption. These programs include Firemin, All Browsers memory zip, Wise Memory Optimizer, SpeedyFox and others. These memory optimization programs work as independent applications and do not thereby control browser memory usage effectively. It is desired that computer applications use little memory and execute faster with a view to allow as many programs to be loaded in the main memory for



execution. With browsers being among such applications, this remains an issue under investigation.

Performance of the reviewed tools highlighted specific challenges while working with them. These weaknesses include, inefficient memory control, poor compatibility issues, overhead to users and decrease in browser performance. An interesting issue was found on the browser reference architecture. The contemporary architecture in use by browsers today, aggravates this problem. The architecture lacks memory control mechanism, which would complement these third party applications. From the weaknesses aforementioned, there was need to relook at the architecture so as to provide a control mechanism for browser memory usage.

## CHAPTER 3

### THE ENHANCED BROWSER REFERENCE ARCHITECTURE

The chapter discusses the structure and behavior of the enhanced browser reference architecture. It discusses the structural components the architectural model adopted in the design of contemporary browsers. The memory analyzer component is integrated as a module in the enhanced architecture.

#### 3.1 Contemporary Browser Architecture

Current browsers adopt the reference architecture for web browsers discussed in section 2.6 of chapter 2. Each module functionality is discussed in the sections herein. An illustration of the interaction of the mentioned modules is as shown in figure 2.8.

##### 3.1.1 User Interface

This module provides the methods with which a user interacts with the Browser Engine. It provides standard web browser features including user preferences, printing functionality, downloading, opening and closing tabs, etc. Browser designers have variant approaches in designing the user interface of the target browser. However, a given browser version depicts slight differences in the user interface from another version of the same type. For instance, earlier versions of Mozilla Firefox had the reload button positioned to the right of the address bar while current versions have it positioned to the left.

##### 3.1.2 Browser Engine

This module provides a high-level interface to the Rendering Engine. It provides methods to initiate the loading of a URL and other high-level browsing actions like reload, back and forward. Furthermore, it provides the User interface with various messages relating to error messages and loading progress. When the browser fails to fetch the content specified by the URL, appropriate messages are conveyed to the User Interface, seeking the intervention of the browser user.

##### 3.1.3 Rendering Engine

This module provides the visual representation of the fetched URL. It comprises various subsystems that enable the browser to interpret the content of the URL. A URL contains two major parts: protocol and web resource. The protocol defines the mechanism through which

resource will be fetched. Common protocols include HTTP and FTP. Web resources include text documents, images/graphics, audio, and video. The multimedia content is interpreted by the appropriate parser to visually human-readable format. A prime component of the Rendering Engine is the HTML parser. The HTML parser is often tightly integrated with the rendering engine for performance reasons and can provide varying levels of support for broken or nonstandard HTML. It can display other types of data via plug-ins or extension; for example, displaying PDF documents using a PDF viewer plug-in. The rendering engine has XML parser subsystem that parses XML data. The JavaScript Interpreter interprets the JavaScript content in the URL. Detailed functionality of mentioned subsystems is discussed in subsections 3.1.3.3 through 3.1.3.5. Different browsers use different rendering engines: Internet Explorer uses Trident, Firefox uses Gecko, and Safari uses WebKit. Google Chrome and Opera Web browsers from version 15 use Blink, a fork of WebKit. The study focussed on Gecko-based browsers whose functionality is described subsections 3.1.3.1 and 3.1.1.2.

### 3.1.3.1 Gecko Rendering engine

This module has the following items that facilitate its functions. Figure 3.1 shows the Gecko subsystems and how they interact with each other (Tali & Paul, 2011a).

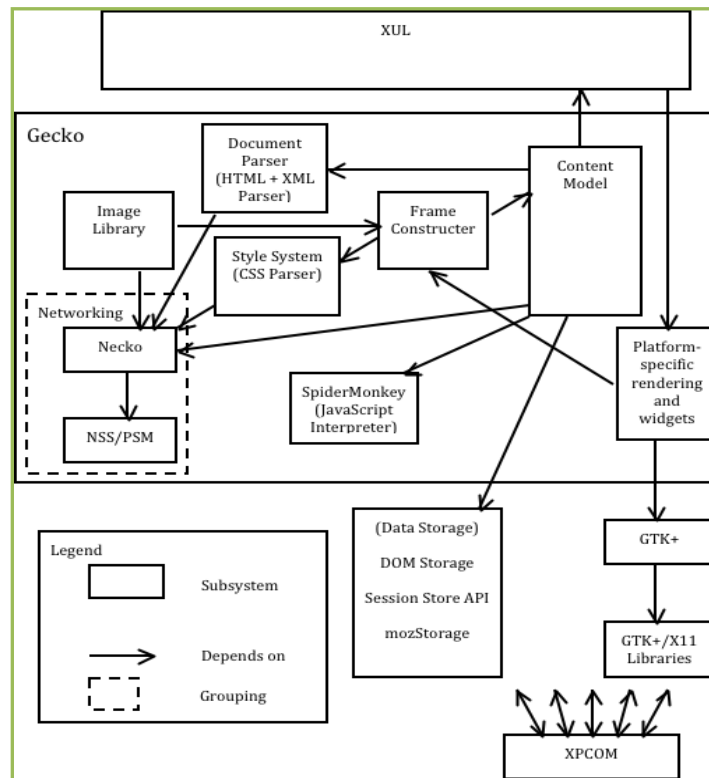


Figure 3.1: The Gecko rendering engine (Tali & Paul, 2011).

- i) Document Parser (HTML & XML Parser)
- ii) Style System: contains the CSS Parser and is responsible for getting the CSS data from Necko and parsing it before sending it to the frame constructor
- iii) Platform-Specific Rendering and Widgets
- iv) Image Library: Interacts with Necko in order to retrieve image data before sending it to the Frame Constructor
- v) Content Model: Interacts with the various components of Gecko, DOM Storage to gather all the data needed before sending it to the frame constructor
- vi) Frame Constructor: Carries out the task of piece together all the information and actually from the rendered web page before sending it back to the UI through the Platform-Specific Rendering subsystem.

### 3.1.3.2 Gecko Rendering Engine : Components functions flow

The Gecko rendering engine components work together to give a visual representation of each content retrieved by the browser engine in a fashion described in four steps as outlined below (Tali & Paul, 2011b). The components are as shown in figure 3.2

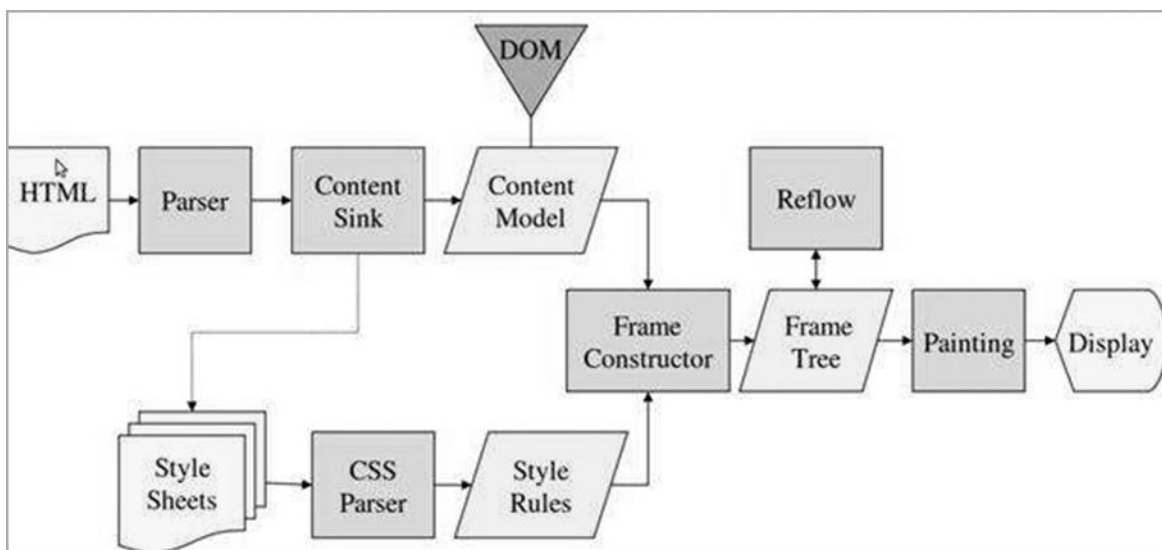


Figure 3.2: Rendering engine-Functions flow diagram (Tali & Paul, 2011b).

**Step 1:** Parsing the HTML document and convert elements to DOM nodes in a tree called the “content tree” – **HTML Parser**

**Step 2:** Parse the style data, both in external CSS files and in style element together with visual instructions in HTML will be used to create another tree, call “render tree” – **CSS Parser**

**Step 3:** After the construction of the render tree it goes through a “layout” process. This means giving each node the exact coordinates where it should appear on the screen

**Step 4:** The next stage is painting—the render tree will be traversed and each node will be painted using the UI backend layer - **Painting**.

### **3.1.3.3 Networking Component**

This component provides the functionality to handle URLs retrieval using the common Internet protocols like HTTP and FTP. It handles all aspects of Internet communication and security; character set translations and Multi-Purpose Internet Mail Extensions (MIME) type resolution. This component may also implement a cache of retrieved documents to minimize network traffic

### **3.1.3.4 JavaScript Interpreter**

This component executes the JavaScript code that is embedded in a URL. Results of the execution are passed to the Rendering Engine for display. The Rendering Engine may disable various actions based on user-defined properties. Where the browser user has set JavaScript code to be disabled, the rendering engine ignores the interpreted material.

### **3.1.3.5 XML Parser**

This is a software library or a package that provides an interface for client applications to work with XML documents. The parser is a generic and reusable component with a standard that has well-defined interface. It checks for proper format of the XML document and may validate the XML documents. Modern day browsers have built-in XML parsers. The goal of a parser is to transform XML data into a human-readable code.

### **3.1.4 Display/UI Backend**

This component is tightly coupled with the host operating system. It provides primitive drawing and windowing methods that are host operating system dependent. Common widgets like combo box, an input box, a checkbox, etc. are drawn using UI properties.

### **3.1.5 Data Persistence**

The Data Persistence component manages the user's data such as bookmarks, cookies, and preferences. The browser may need to save all sorts of data locally. Browsers also support storage mechanisms such as localStorage, IndexedDB, WebSQL and FileSystem (Michael Coates, 2010).

### 3.2 The Enhanced Browser Architecture

The enhanced architecture incorporates a Memory Analyzer component as shown in figure 3.3. The memory analyzer component interacts with the operating system to track memory usage in real-time and to check browser memory consumption against the set threshold total memory. After analysis, the user is provided with possible actions to take to prevent memory hogging. Consequently, more applications can be run from the system. This guarantees that browsers do not make computer to freeze by delimiting other legitimate applications from running. As a result, it improves the level of multiprogramming and ultimately improves the user-browsing experience. The analyzer is implemented as a software module included in the web browser application.

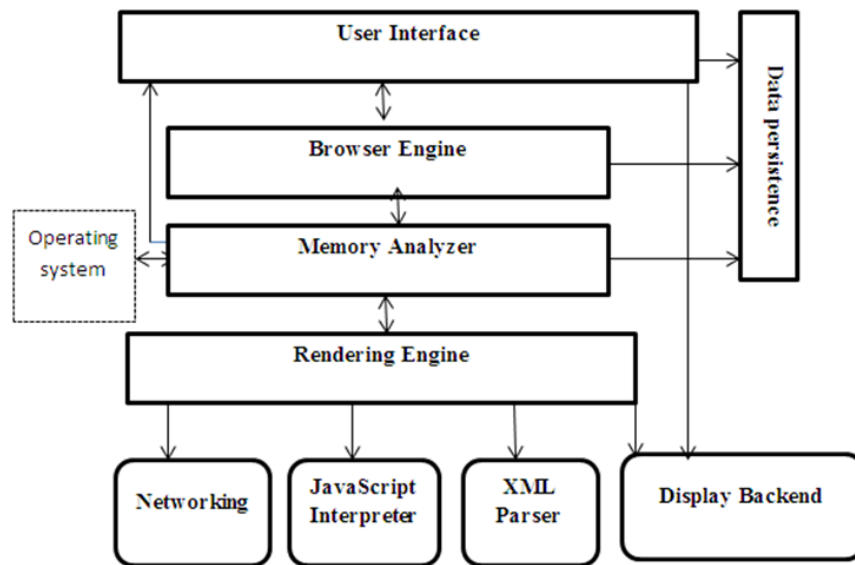


Figure 3.3: The enhanced browser architecture(Source: Research)

#### 3.2.1 Memory Analyzer

This component checks real-time memory consumption for the browser against the threshold total memory limit set by the user and gives feedback information to the user on possible actions to take in order to prevent memory hogging by the browser. Memory analysis is done after the browser engine has retrieved a resource. The rendering engine interprets and gives a visual representation of the URL with the help of parsers and JavaScript interpreter if memory space is available. It was envisioned that this component would provide a memory control mechanism that would hence control memory hogging. Furthermore, the analyzer provides a garbage collection mechanism to reclaim unused memory from the browser objects.

### 3.2.2 Flow diagram of memory analyzer

A conceptualized design of a Memory Analyzer and its interactions with other modules is as shown in figure 3.4. When a user enters a URL on the browser's address bar and hits the Go button, the Browser Engine takes the URL and attempts to fetch its content. The Memory Analyzer performs analysis of the memory consumed against the threshold memory as set by the user. If the memory is lower than the threshold memory, it passes the content of the URL to the rendering engine for further actions. However, if the consumed memory gets higher than the threshold memory, a notification error message is passed to the higher modules for action to be taken by the user.

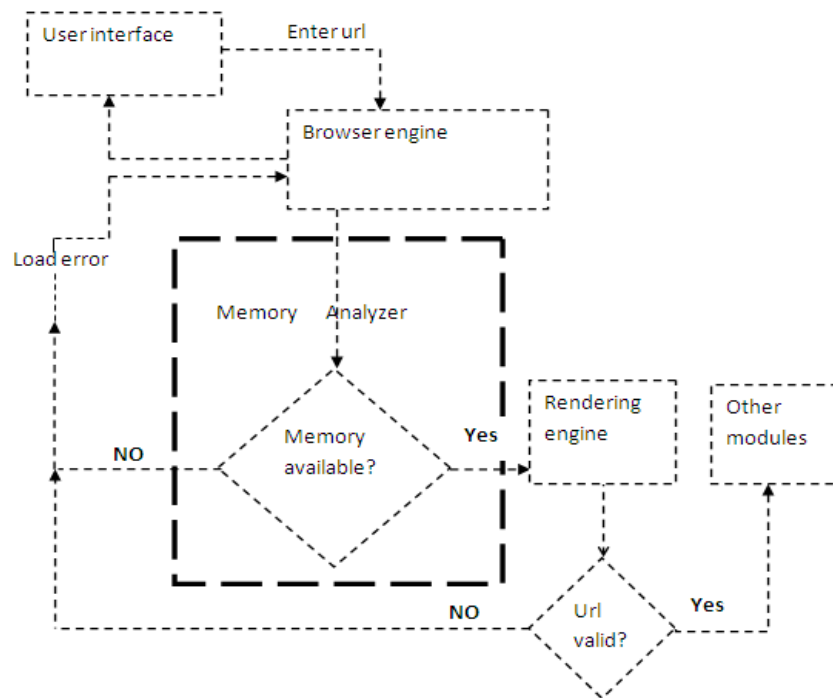


Figure 3.4: The Flow diagram of a memory analyzer(Source: Research)

### 3.3 Summary

Memory hogging is a habitual phenomenon in browser applications in computers with limited memory. In an attempt to solve this problem, the memory analyzer was integrated to the current model to play the role played by third-party applications discussed in chapter two and notify the user when memory hogging is detected. The researcher designed a browser prototype and integrated the memory analyzer in it. A detailed report on the design and implementation of the new strategy is discussed in chapter 4.

## **CHAPTER 4**

### **METHODOLOGY**

The chapter discusses the processes involved in realizing the research objectives. It begins by outlining the approach adopted by the researcher in designing and developing a browser prototype which incorporates a memory analyzer in its architecture. Section 4.1 highlights the adopted research design. Section 4.2 through 4.3 discusses the software development process where the techniques and development tools used in development of the browser prototype and memory analyzer are discussed. Section 4.4 through section 4.6 discusses the research method and data collection tools used for evaluation of the developed prototype. Systematic experiments are carried out to attest the necessity of the memory analyzer in the proposed browser architecture.

#### **4.1 Research Design**

The study adopted experimental research design with a view to assessing the memory consumption for the selected browsers on various websites running on Windows operating system. This design was appropriate for this research since it provided a basis for evaluating the efficacy of integrating the memory analyzer in the browser reference architecture. Target browser was Mozilla Firefox. This target was adopted based on its global market share and the fact that its code is open source.

#### **4.2 Development of Browser Prototype**

Prototyping methodology was adopted for this study. A browser prototype based on the Mozilla Firefox project was used to simulate how the proposed architecture works. Visual Studio 2010 was used as the development platform. The choice of this platform was informed by its integrated environment for developing windows applications. The prototype was coded using C Sharp (C#) language. This choice was preferred based on the language used in coding GeckoFx wrapper class. The GeckoFx package with a corresponding Extended User Language (XUL) runtime for a particular version of the Mozilla Firefox browser is readily available from the Bitbucket repository. This provided an enabling environment for reuse and integration of the Gecko rendering engine in the developed browser prototype. Visual studio 2010 software suite was the adopted Integrated Development Environment (IDE) for realizing the prototype.



### **4.2.1 Prototyping Model**

Prototyping technique can be used in developing a large and complex system. A browser is among such software systems. The study adopted this technique with a view to providing a comparative study in terms of performance during its evaluation. This technique was used in both requirements phase and the design phase to demonstrate a concept and options such as interfaces and technology to be used (Pawel & Marcin, 2015).

Rapid Architected Analysis (RAA) was adopted in the requirements phase. This approach attempts to derive system models from existing systems or discovery prototypes (Xi Yang, et al., 2017). The rapid architected analysis employs reverse engineering techniques to unveil the components of a system (Chua, Leong & Lim, 2010). It is a process of discovering the technological principles of a device, object, or system through analysis of its structure, function, and operation (Elda, 2007). Software reuse technique was also adopted in developing the prototype. The main component that was reused in this process was the rendering engine. The Gecko rendering engine already implemented in C# was integrated with other browser modules as provided by the browser control object in the Visual Studio.

### **4.2.2 Development Procedure**

The study developed a browser prototype with a memory analyzer component incorporated into it. Microsoft Visual Studio 2010 was used as a development environment and all the components were coded in C# language.

### **4.2.3 Architectural Model**

The browser prototype was anchored on the reference browser architecture as postulated by Allan and Michael (2006). The model structure and detailed description of its functionality is given in section 3.1. Current browsers have referred to the architectural model in designing and developing their web browsers. This acted as a benchmark in developing the browser prototype.

### **4.2.4 Design of the Software System**

The architectural design of the target system was modularized with a view to providing manageable units. Each module was implemented separately and later merged with other modules with a view to having complete the prototype.

#### **4.2.4.1 Browser prototype**

A development environment for a general windows application was set in the visual studio. Visual Studio browser control object was used to provide a managed wrapper for the web browser ActiveX control, which duplicates Internet Explorer Web browsing functionality. However, the proposed approach focused on Mozilla based browser, which would provide a reasonable testing environment with Mozilla Firefox as a controlled browser during the experiments. This necessitated a change of the target rendering engine. Mozilla based browsers use Gecko as the rendering engine. The study, therefore, integrated GeckoFx wrapper class as an assembly in Visual Studio project. A XUL Runner runtime package matching the target version of Mozilla Firefox was also included as a dependency to the GeckoFx-core assembly file.

#### **4.2.4.2 Embedding Gecko Rendering Engine**

To embed the Gecko rendering engine in a Windows browser application, the following components were required:

- i. XulRunner: XULRunner is a Mozilla runtime package that can be used to bootstrap XUL+XPCOM applications that are as rich as Firefox and Thunderbird. It provides mechanisms for installing, upgrading, and uninstalling these applications. This component was downloaded from <http://ftp.mozilla.org/pub/mozilla.org/xulrunner/>
- ii. GeckoFx: GeckoFx is a .NET assembly file, which contains the Gecko rendering engine. The chosen version must match with the XulRunner version. The research used version 33.0, which was downloaded from <https://bitbucket.org/geckofx>.

#### **4.2.4.3 Embedding Procedure**

The procedure was done in seven steps as outlined herein.

- i. Download the GeckoFx assembly file and unpack it to extract the files as shown in figure 4.1. There are two assembly files that are important in the target directory. These are Geckofx-Core.dll and Geckofx-winforms.dll. Geckofx-Core.dll provides an implementation of the core functions of the components that work in collaboration to render the content fetched by the browser engine. Geckofx-winforms.dll provides methods for handling browser events.

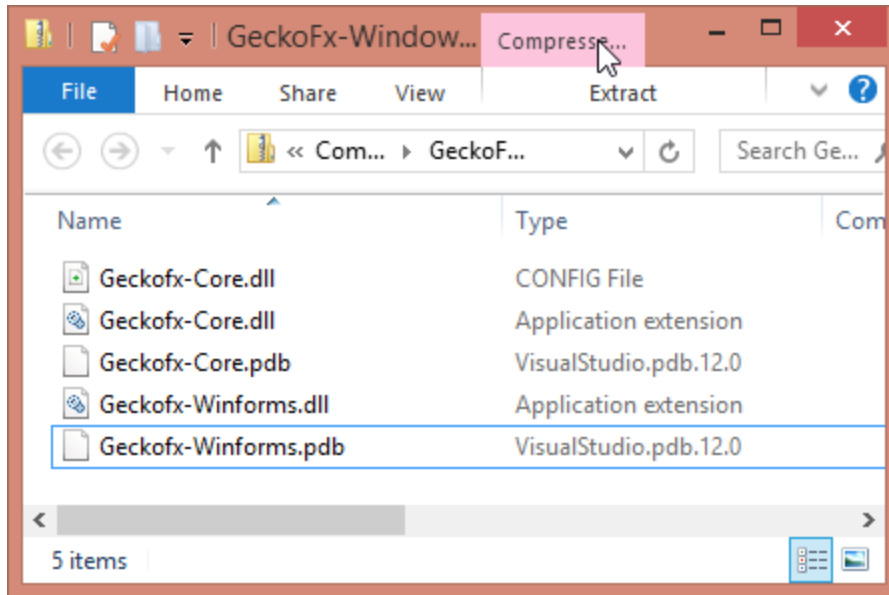


Figure 4.1: Unpacking GeckoFx package (Source: Research)

- ii. Add references of the assembly files to the browser project by clicking browse and selecting the Geckofx-Core.dll and Geckofx-Winforms.dll as shown in figure 4.2.

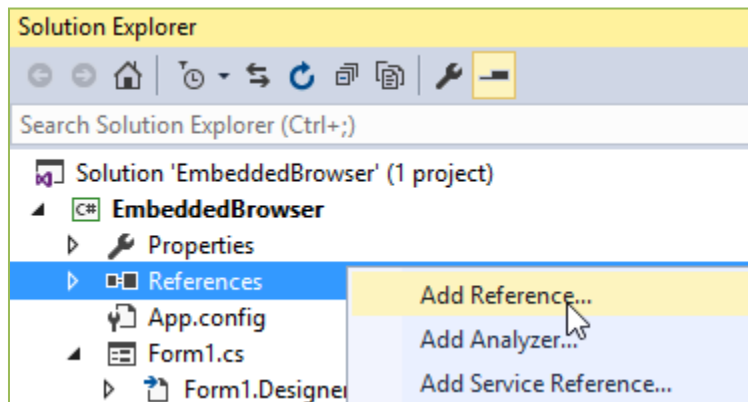


Figure 4.2: Adding GeckoFx assembly files as references (Source: Research)

- iii. Select the respective assembly files and click Add button to add them to the project as shown in figure 4.3.

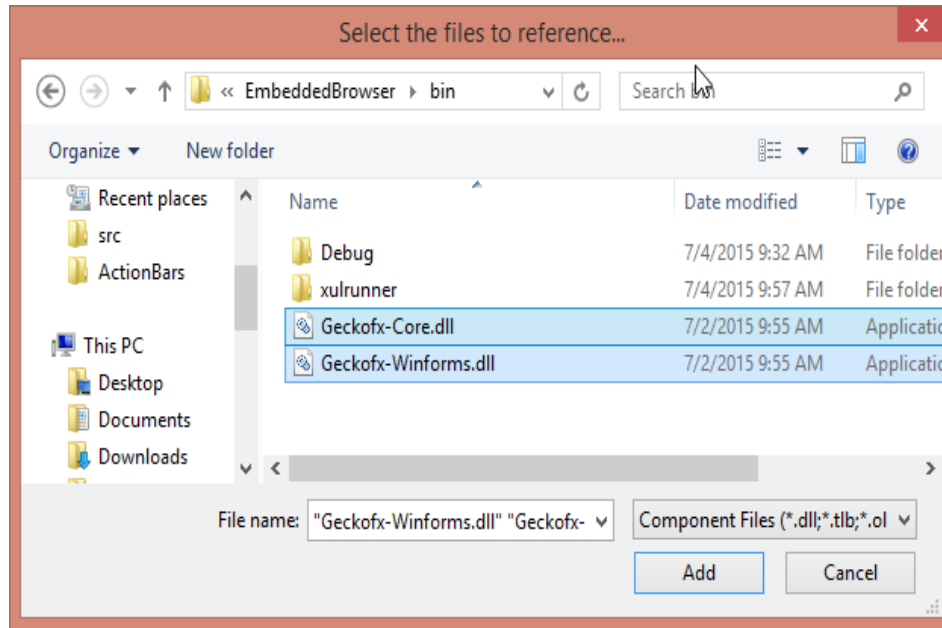


Figure 4.3: Selecting GeckoFx assembly files (Source: Research)

- iv. In the toolbox, right-click, and then select “Choose Item”, select Geckofx-Winforms.dll, and the Gecko winform control will pop up in the toolbox as shown in figure 4.4

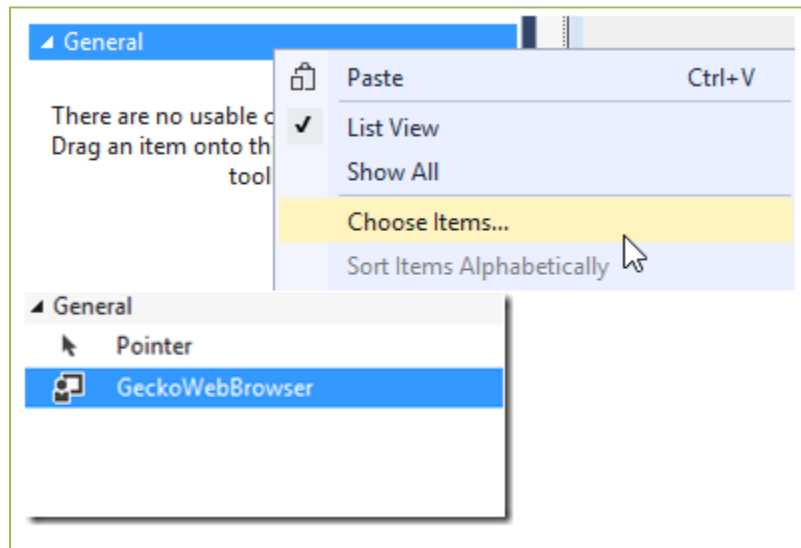


Figure 4.4: Adding GeckoFx browser control to the toolbox (Source: Research)

- v. Drag a GeckoWebBrowser control to the winform designer, and name it “embeddedBrowser.” The windows form adds browser control object, which indicates the version of Mozilla Firefox browser it implements as shown in figure 4.5.

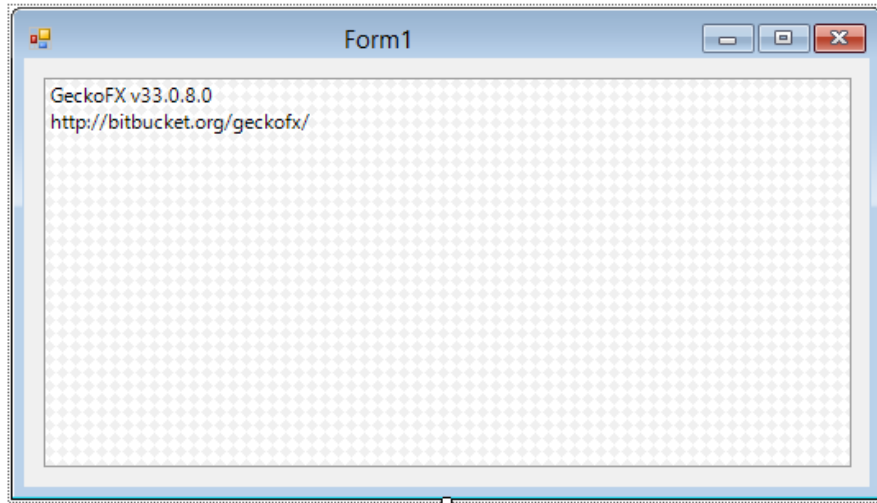


Figure 4.5: Adding GeckoFx browser control to a windows form (Source: Research)

- vi. To change the rendering engine used by the browser control, the Gecko class has to be imported in the program. As indicated section 4.2.4.2, the Gecko engine depends on Xpcom components shipped in the xulrunner package. In the form1.cs file, import the Gecko class and reference the Xpcom component file path as shown in figure 4.6.

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Gecko;

namespace EmbeddedBrowser
{
    public partial class Form1: Form
    {
        public Form1()
        {
            InitializeComponent();
            Xpcom.Initialize(@"..\xulrunner");
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            geckoWebBrowser1.Navigate("http://www.google.co.ke");
        }
    }
}

```

Figure 4.6 GeckoFx browser prototype code-snippet (Source: Research)

- vii. The line `Gecko.Xpcom.Initialize(@"..\xulrunner");` specifies where the xulrunner runtime is located. In this case, it is put into a folder (`@"..\xulrunner"`). To demonstrate how the prototype works, the navigate method call is made with the URL as a parameter.

The browser engine fetches the content of the specified URL and invokes the components in the Gecko rendering to give its visual representation. To achieve this, the application needs to be built and run using the Visual Studio build tools. After running the application, a visual representation of the URL passed in the browser using the `'geckoWebBrowser1.navigate'` method shown in figure 4.6 is depicted in figure 4.7

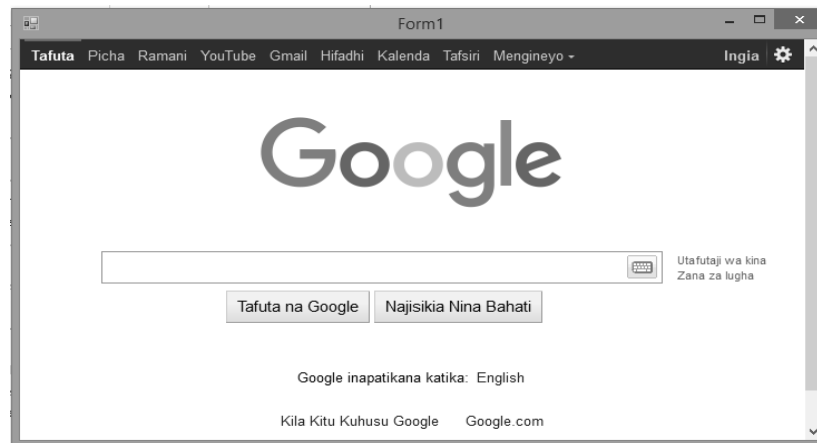


Figure 4.7: GeckoFx browser prototype in action (Source: Research)

### 4.3 Memory Analyzer

This component checks real-time memory consumption for the browser against the threshold total memory limit set by the user and gives feedback information to the user on possible actions to take to prevent memory hogging by the browser. Memory analysis is done after the browser engine has retrieved a resource. The rendering engine interprets and gives a visual representation of the URL with the help of parsers and JavaScript interpreter if memory space is available.

#### 4.3.1 Memory Analyzer Interface Design

To demonstrate the proposed approach, the researcher designed the memory analyzer interface with a view to showing how memory settings would be made by the user, computation of available memory and consumed memory by the browser process. Figure 4.8 shows the interface design of the memory analyzer. In the diagram, the labels attached to total physical RAM, available memory, threshold memory, MOB memory consumption, and Mozilla Firefox buttons, are updated upon execution of the analyzer logic. MOB is the name of the newly developed browser prototype.

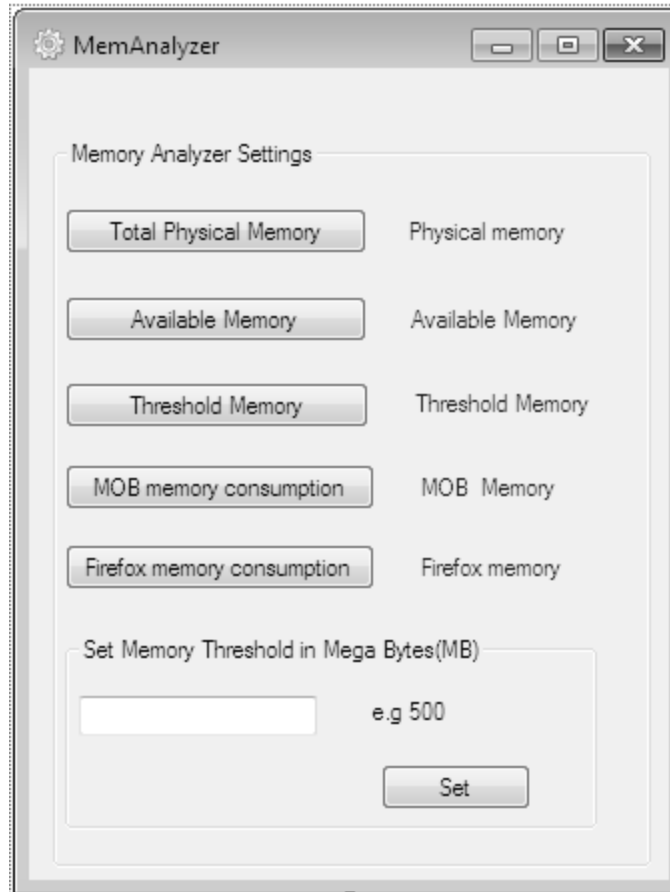


Figure 4.8: Memory analyzer interface design (Source: Research)

#### 4.3.2: Setting Memory Threshold

The maximum amount of memory the user desires the browser to consume is set using textbox control and the setting logic implemented when the user clicks the set button. Success or fail message is displayed to confirm the event was fired. The analyzer checks the memory threshold value from browser system properties and uses it as a reference. However, the user changes the default value depending on the size of RAM in the host computer. Validation is done to guarantee that the input value does not exceed the value of the memory available for use in the entire system. Upon successful memory setting, the browser adapts to the new change accordingly. Any time the browser memory consumption reaches the set memory threshold, a notification is made to the user that the browser is hogging memory.

The values displayed besides the memory settings-group box in figure 4.9, demonstrates the current memory readings fetched from the operating system. The input box takes in the desired value for computation. The browser user makes a valid setting if the specified value is less than

the available memory. However, if the value specified is higher than the available memory, the validation function returns false.

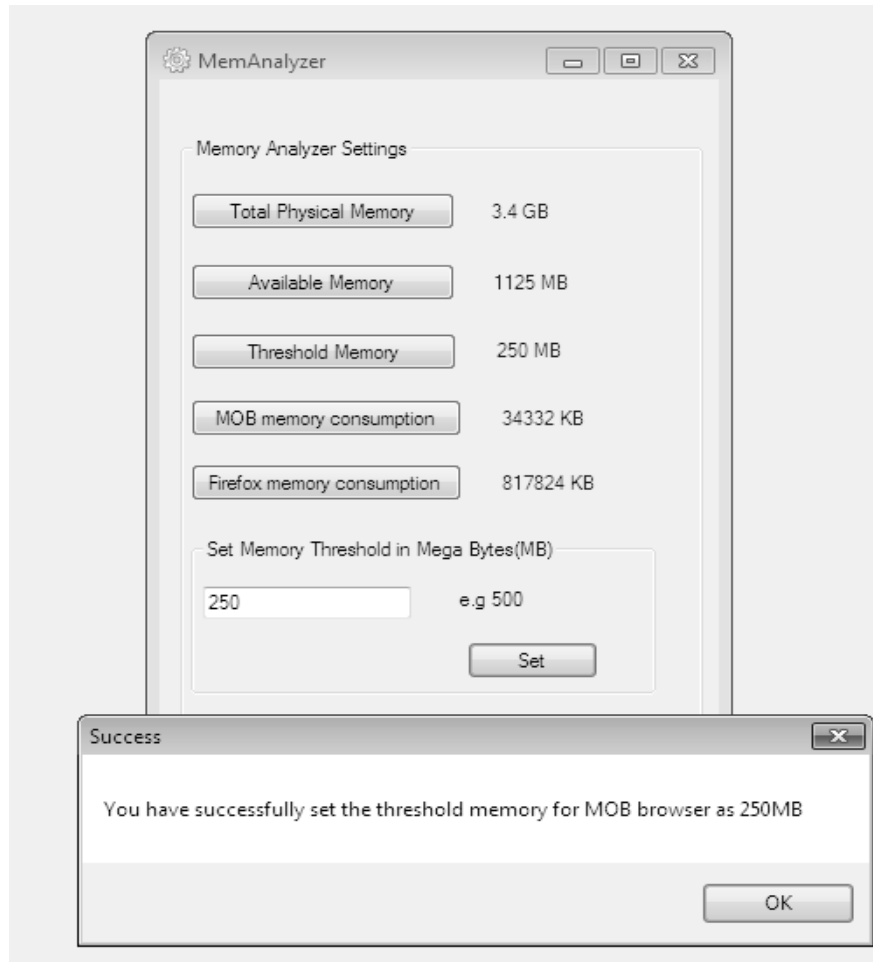


Figure 4.9: Setting memory in the memory analyzer (Source: Research)

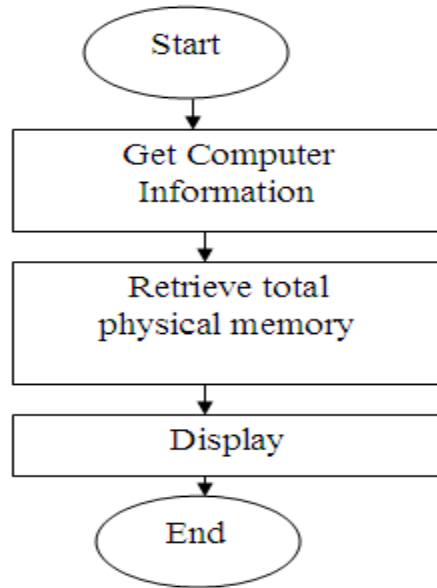
### 4.3.3: Memory Computation Logic

To carry out memory analysis, memory computation was inevitable. Total physical memory, available memory, and consumed memory values were computed using the generic Windows routines as in the case of Windows Task Manager.

#### 4.3.3.1 :Physical Memory Computation Logic

The flow chart in figure 4.10 describes the logic function for computing physical memory. The function returns the total amount of physical memory available in a host computer.





*Figure 4.10: Physical memory computation logic (Source: Research)*

#### **4.3.3.2: Available Memory Logic**

This function computes the available memory for use by computer programs. It is not equal to the physical capacity of the installed memory but is much lower. It constitutes to the free memory that is available for allocation by the operating system for any process that would be scheduled for execution. For any given process to be executed, the operating system must load it in the main memory. In a multiprogramming environment, this memory space is shared among all processes that have been scheduled to run by the operating system. Figure 4.11 illustrates the memory layout in an environment where there are several programs ready to run. The empty space in the diagram corresponds to the available memory space. The jobs represent the programs in the main memory.



Figure 4.11: Memory Layout in single processor system (Source: Research)

### 4.3.3.3: Browser Memory Consumption

The function calculates the amount of memory space the active process has consumed. The logic implementation is described by the figure 4.12. In this context, the value fluctuates since browser memory consumption is dynamic.

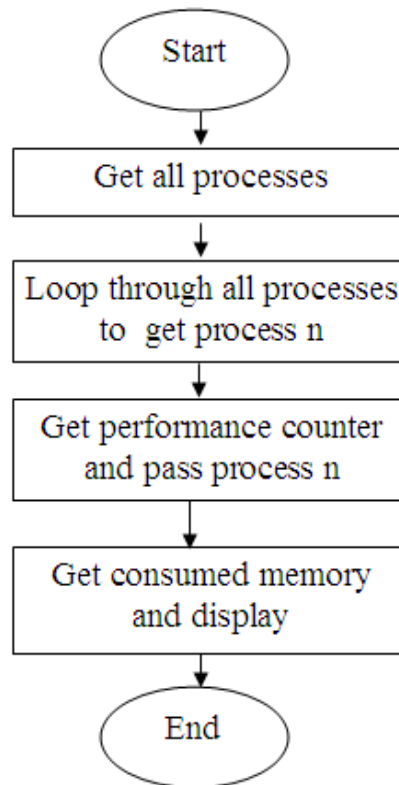


Figure 4.12: Browser Memory consumption computation logic (Source: Research)

As the browser starts, memory consumption rises exponentially and then stabilizes after the content of the URL has been fully rendered. In a dynamic webpage, memory consumption may fluctuate persistently depending on background activities. However, this phenomenon is different when a static web page is loaded.

#### **4.3.3.4: Memory computation events timer**

To capture memory consumption readings, the computation logic is executed after every 5 seconds. This necessitates a timer function. Computation logic for available memory and browser memory consumption is executed when the timer event is fired. In this fashion, memory readings are recorded for later analysis.

### **4. 4 Experimental Setup**

The study evaluated Mozilla-based browser prototype based on GeckoFx version 33.0 with an integrated memory analyzer and a contemporary Mozilla Firefox version 33. The developed prototype was christened MOB. For each experiment, each browser was tested individually in a single target computer configured to run a Windows operating system of 32-bit architecture. Selected sites like electronic mail service providers (Gmail.com), social networks (Facebook.com), entertainment (Youtube.com) and search engines (Google.com) were investigated. Selection of these sites was based on the global market share (SimilarWeb, 2018). Memory consumption by the developed prototype and Mozilla Firefox browsers were compared in ten sets each in parallel. Memory consumption by both browsers was investigated on both homogeneous and heterogeneous website tabs. This was aimed at determining the variance in memory consumption while opening browser tabs for various websites. Memory consumption readings were done after the browser finished loading the content.

#### **4.4.1 Evaluation Metrics**

A number of metrics were adopted during the investigation. These included: available memory, consumed memory, and threshold memory.

- i. **Available Memory(AM).** This value represented the total available memory for use by all programmes. It is lower than the actual value of installed RAM. AM determines the number of programs that can comfortably get loaded in the main memory ready for execution. This translates to the degree of multiprogramming. A computer system with a larger size of AM can accommodate a higher number of programs in the main memory

compared to one with less. The research evaluated the performance of the two browsers on 1 GB RAM, machine.

- ii. **Consumed Memory(CM):** This represented the memory demand posed by the browser to render a given URL. The value of CM was evaluated in both browsers running in parallel. The research evaluated how CM, directly and indirectly, affected AM.
- iii. **Threshold Memory(TM):** The capped memory in which the browser should not exceed. This value was varied from 100 MB at an interval of 100 MB. TM indicates the highest amount of memory the browser process is allowed to consume. The moment CM supersedes TM; a red flag is raised to give a warning of memory hogging.

#### **4.5 Population, Sample and Sampling Procedure**

The research focused on Mozilla Firefox project. Cluster sampling method was used to get a representation of websites from which to obtain the data based on similar global web rankings (SimilarWeb, 2018). Websites were categorized into social network, search engine, TV and video and Email. Under Email, Google mail was investigated. In social networks category, Facebook was investigated. In third category, entertainment, YouTube was the put into focus. For search engines, Google was investigated.

#### **4.6 Data Analysis**

A quantitative data analysis procedure was adopted where descriptive statistics such as frequencies, percentages, means, and standard deviations were used to summarize data. Tables and figures were used to illustrate summarized data. Results from analyzed data were used to affirm the applicability of a memory analyzer in its effect in controlling memory hogging and inferences were drawn based on the results. Inferential statistics were used to deduce the general effect of integration of the memory analyzer to browser memory consumption. Linear regression model was used to assess the relationship between consumed memory and number of browser tabs. T-test was used to assess the statistical difference between memory consumption by MOB and Mozilla Firefox browsers.

#### **4.7 Summary**

This chapter gave a detailed account of the software development techniques and tools used to design the browser prototype. It highlighted the implementation techniques and described how the memory analyzer was integrated in the developed prototype. Secondly, the chapter discussed

the research design and methods/techniques used to collect as well as analyze the data about the research. It also provided an overview of how the research was carried out and how the data collected were used to generate results. The research methodology adopted was experimental. This method was adopted with a view to assessing the memory consumption for the selected browser on various websites running on Windows operating system. With this prototype model, it was possible to test the efficacy of the memory analyzer in controlling memory hogging and subsequently provided a reasonable ground for testing the hypothesis that, there is no difference in memory consumption by analyzer-integrated browser and non-analyzer integrated browser. The prototype further allowed the study to evaluate the performance and effectiveness of the integration of memory analyzer module before actual adoption. Presentation of results is discussed in detail in chapter 5.

## CHAPTER 5

### RESULTS AND DISCUSSION

This chapter presents the results of the experiments that were conducted. Section 5.1 presents experimental results, analysis, and discussion while section 5.2 discusses the hypothesis test. Lastly, section 5.3 gives a conclusion about the findings.

#### 5.1 Presentation of results

Results are presented based on data obtained from tests as described in section 4.3.

##### 5.1.1: Memory consumption by default processes

The processes shown in table 5.1 characterized the testing environment. The researcher launched the browser processes only for the entire testing period. Several computer users owned these processes in the system including System, Test (Logged in user), Network service, and local service. Memory allocation is given in Kilobytes, abbreviated as “K”.

*Table 5.1: Memory consumption by default processes*

<b>Applications/Processes</b>				
<b>Name</b>	<b>Program file</b>	<b>Instance</b>	<b>Owner</b>	<b>Memory Allocation</b>
Client server Runtime process	Csrss.exe	1	System	752K
	Csrss.exe	2	System	796K
Desktop Window manager	Dwm.exe	1	Test	15,392K
Windows explorer	Explorer.exe		Test	20,600K
Local security authority process	Isaas.exe	1	System	1,860K
Local session manager service	ism.exe	1	System	616K
Microsoft windows search indexer	searchIndexer.exe	1	System	5,088K
Services and controller app	Services.exe	1	System	1940K
Windows session manager	smss.exe	1	System	128K
Snipping tool	snippingTool.exe	1	Test	1,368K
Spooler subsystem app	Spoolsv.exe	1	System	1,032K
Host process for windows services	Svchost.exe	1	System	1,496K
	Svchost.exe	2	Network service	1,592K
	Svchost.exe	3	System	2,1036K
	Svchost.exe	4	System	8,480K
	Svchost.exe	5	Network service	2,708K
	Svchost.exe	6	Local service	2,476K
	Svchost.exe	7	Local service	1,624K
	Svchost.exe	8	Local service	944K

	Svchost.exe	9	Local service	4,096K
	Svchost.exe	10	System	1,708K
system	system	1	System	40K
Percentage of time system is idle	System idle process	1	System	24K
host process for Windows tasks	Taskhost.exe	1	Test	1,088K
Windows task manager	Taskmgr.exe	1	Test	1,840K
WMI provider host	WmPrvSE.exe	1	Network service	1,572K
Windows driver foundation	WUDFHost.exe	1	Local service	484K
Windows logon application	Winlogon.exe	1	System	452K
Windows start-up application	Wininit.exe	1	System	384K

(Source: Research)

### 5.1.2 Browser memory consumption

Mozilla Firefox and MOB memory consumption were critically investigated for analysis. Memory consumed by launching a single tab for each website category in both browsers was recorded in table 5.2.

Table 5.2: Mozilla Firefox and MOB memory consumption

Applications/Processes					
Browser Name	Program file	Instance	Owner	Memory Allocation	
Mozilla Firefox	Firefox.exe	1	Test	On google	74,443K
				On youtube	183,628K
				On facebook	257,608K
				On gmail	262,052K
MOB	MOB.exe	1	Test	On google	33,330K
				On youtube	130,615K
				On facebook	160,628K
				On gmail	210,012K

(Source: Research)

The results in table 5.2 indicate that the browser processes were the highest memory consumers amidst other processes shown in table 5.1. This characteristic depicts the browser as memory ravenous application. The browser consumes a minimum of 33.3 MB on a single Google tab while windows' explorer consumes the highest among the rest with a value of 20.6 MB. Interestingly, MOB browser consumes less memory compared to Mozilla Firefox in the four tested websites.

### 5.1.3 Memory consumption averages by MOB and Mozilla Firefox with homogeneous website tabs

Table 5.3 illustrates MOB and Mozilla Firefox memory consumption averages across the four tested websites. X represents computer freeze. Memory consumption by both browsers varied

with the number of opened tabs and with the content loaded in tabs. Memory consumption increased with the number of tabs open. Results indicate that memory increased gradually while accessing Google, YouTube, Facebook, and Gmail respectively in both browsers. To perform an online search using Google required an average of 33.3 MB and 74.4 MB on MOB and Mozilla Firefox respectively in a single tab. This value gradually increased as more tabs were opened. To give a visual representation of the searched content by Google search engine, which comprises mainly text required relatively low memory compared to other investigated website categories. Access to YouTube page, which comprises mainly video content, required relatively high memory in comparison to loading text. An average of 130.6 MB and 183.6 MB on MOB and Mozilla Firefox respectively were required to load content on a single tab and this value increased exponentially as numbers of tabs were increased. Although content on Facebook, comprise mainly text and images, access to these pages revealed debatable results. Access to Facebook required 160.6 MB and 257.6 MB on MOB and Firefox respectively, which is relatively higher memory than access to YouTube. Background processes on Facebook were characterized by timed events that executed severally hence requiring more memory.

Table 5.3: MOB and Mozilla Firefox memory consumption averages with homogenous website tabs

	Website/ Memory (MB)	Tabs										
		1	2	3	4	5	6	7	8	9	10	
MOB	Google	CM	33.3	38.4	42.5	49.2	53.6	56.0	59.8	65.6	70.9	73.1
		AM	620.4	609.8	606.5	597.6	594.8	591.2	588.4	586.0	580.3	578.8
	YouTube	CM	130.6	190.3	277.0	326.7	394.2	432.2	494.9	550.4	612.3	X
		AM	523.7	483.6	412.9	320.3	275	229.2	182.6	122.4	75.3	X
	Facebook	CM	160.6	236.5	365.6	448.1	515.3	579.7	640.8	694.4	X	X
		AM	451.1	387.7	312.5	227.5	142.6	112.8	91.7	58.8	X	X
Gmail	CM	210.0	344.7	462.9	584.3	628.5	678.1	698.3	X	X	X	
	AM	344.4	310.1	214.0	110.5	92.2	79.4	49.6	X	X	X	
Mozilla Firefox	Google	CM	74.4	90.0	97.0	103.6	110.8	120.1	127.6	133.9	144.0	152.0
		AM	540.8	528.0	518.8	515.3	508.7	500.9	492.8	486.7	474.9	465.8
	YouTube	CM	183.6	225.5	292.4	354.7	410.2	473.2	534.9	593.2	628.4	X
		AM	492.5	443.6	380.6	282.4	239.5	198.2	137.6	96.6	44.5	X
	Facebook	CM	257.6	333.5	414.4	483.8	546.3	612.4	680.8	714.4	X	X
		AM	382.3	306.7	237.5	174.5	125.6	93.8	71.7	35.3	X	X
Gmail	CM	262.0	384.7	487.9	606.3	656.2	696.1	X	X	X	X	
	AM	324.9	289.1	198.5	93.8	63.2	46.3	X	X	X	X	

(Source: Research)

On average, to load a single tab on Gmail required 210 MB and 262 MB on MOB and Mozilla Firefox respectively. A similar phenomenon witnessed by access to Gmail was attributed to a series of background processes that execute sequentially to realize the Gmail application



functionality. Access to both Facebook and Gmail posed increased memory consumption as the number of tabs were increased for both browsers.

Varying TM values guaranteed that MOB would consume not more than the set memory value thus controlling memory hogging. This phenomenon provided substantive value for AM, which provided for more programs in the main memory thus enhancing concurrency. In Mozilla Firefox, regulating how much memory it would consume was not possible as the model in which it is built-on, lacks memory analyzer. The study established that AM gradually reduces with an increase in CM. When the available memory reduces below 70 MB, the computer starts to crawl and eventually freezes.

Computer froze while opening beyond seven, eight, and nine tabs for Gmail, Facebook, and YouTube respectively on MOB browser. However, the same phenomenon was witnessed while opening six, eight, and nine tabs for Gmail, Facebook, and YouTube respectively on Mozilla Firefox. The CM and AM values do not add to a constant since available memory is dynamically modified by operating system memory management routines.

Figure 5.1 through Figure 5.4 shows how available memory and consumed memory varies by opening various homogenous website tabs. Analysis and discussion of results obtained are discussed in subsections herein.

#### ***5.1.3.1 Variation of Available and Consumed memories with Google tabs***

Consumed memory increases linearly with the number of tabs. In MOB, consumed memory increases by 4.45 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $C = 4.457t + 29.727$ .

In Mozilla Firefox, consumed memory increases by increases by 8.13 MB relative to 1 unit change in a browser tab. Regression model is expressed as  $C = 8.1345t + 70.6$ . The value  $C$  in the equations represents consumed memory and  $t$  represents the number of tabs. In both browsers, memory consumption is directly proportional to the number of tabs with a high prediction of 99.36% and 99.33% on MOB and Mozilla Firefox respectively. The study established that there is a statistically significant association between browser tabs and consumed memory. Additionally, the difference in memory consumption by both browsers increases linearly with an increase in the number of tabs as expressed by  $Y_c = 3.677x + 40.87$  where  $Y_c$  represents consumed memory difference and  $x$  represents the number of tabs.

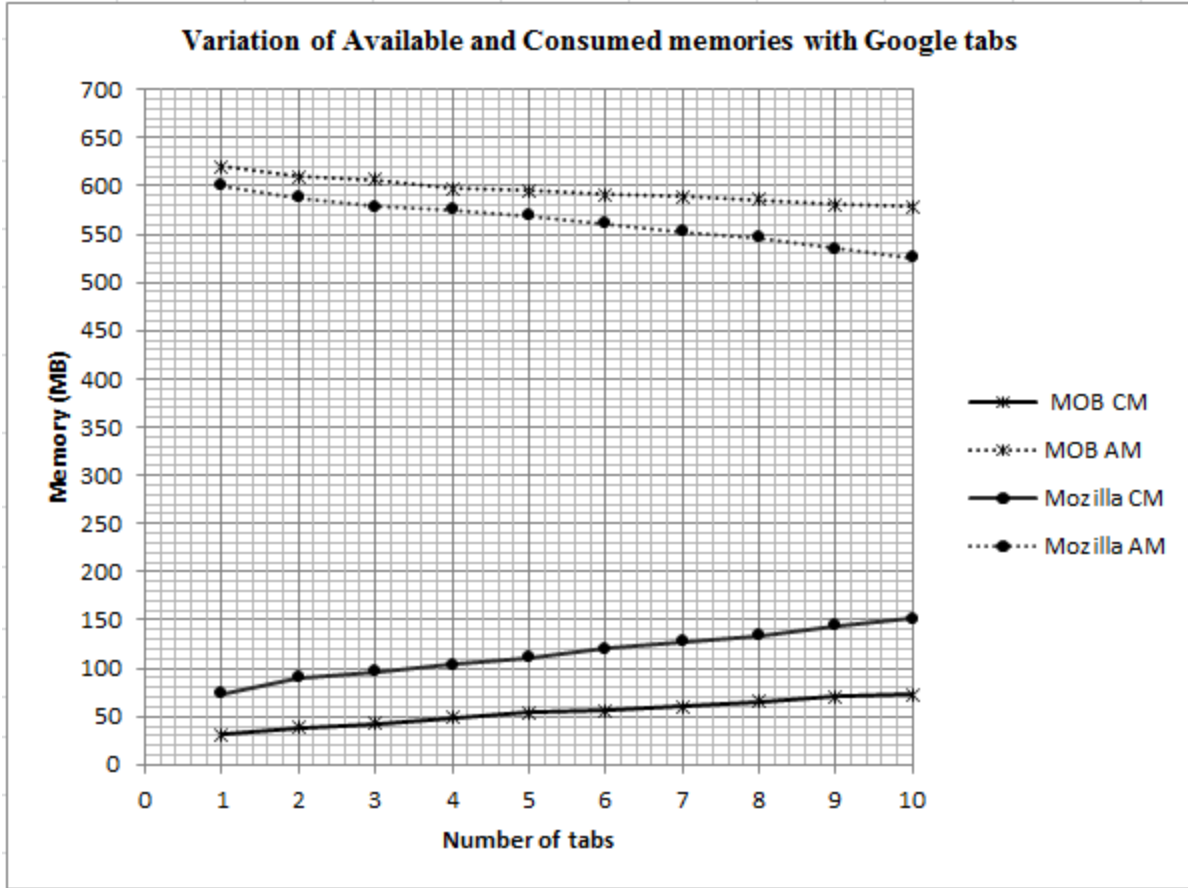


Figure 4.1: Variation of Available and Consumed memories with Google tabs (Source: Research)

Available memory decreases linearly with an increase in the number of tabs. In MOB, available memory decreases by 4.33 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $A = -4.331t + 619.2$ . In Mozilla Firefox, available memory decreases by 7.77 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $A = -7.7727t + 546.02$ . The value  $A$  in the equations represents available memory and  $t$  represents the number of tabs. In both browsers, available memory is inversely proportional to the number of tabs with a high prediction of 95.56% and 99.01% on MOB and Mozilla Firefox respectively. Additionally, the differences in available memory while both browsers are open increases linearly with an increase in number of tabs as expressed by  $Y_a = 4.017x + 76.88$  where  $Y_a$  represents available memory difference and  $x$  represents the number of tabs.

Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is  $1.38E-52$ , which is, less than

the chosen  $\alpha$  hence there is a statistical difference in memory consumption between MOB and Mozilla Firefox on Google. Mean difference in memory consumption is 53.93 MB.

### ***5.1.3.2 Variation of Available and Consumed memories with YouTube tabs***

Consumed memory increases linearly with number of tabs. In MOB, consumed memory increases by 59.14 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $C = 59.14t + 83.033$ . In Mozilla Firefox, consumed memory increases by 58.10 MB relative to 1 unit change in a browser tab. Regression model is expressed as  $C = 58.097t + 120.19$ . The value  $C$  in the equations represents consumed memory and  $t$  represents the number of tabs. In both browsers, memory consumption is directly proportional to the number of tabs with a high prediction of 99.57% and 99.74% on MOB and Mozilla Firefox respectively. The study established that there is a statistically significant association between browser tabs and consumed memory. Available memory decreases linearly with an increase in the number of tabs. In MOB, available memory decreases by 57.15 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $A = -57.148t + 577.41$ .

In Mozilla Firefox, available memory decreases by 56.72 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $A = -56.72t + 540.88$ . The value  $A$  in the equations represents available memory and  $t$  represents the number of tabs. In both browsers, available memory is inversely proportional to the number of tabs with a high prediction of 99.10% and 98.94% on MOB and Mozilla Firefox respectively. Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is 0.1589, which is greater than the chosen  $\alpha$  hence there is no statistical difference in memory consumption between MOB and Mozilla Firefox on Google. Mean difference in memory consumption is 32.14 MB.

Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is 0.1589, which is greater than the chosen  $\alpha$  hence there is no statistical difference in memory consumption between MOB and Mozilla Firefox on Google. Mean difference in memory consumption is 32.14 MB

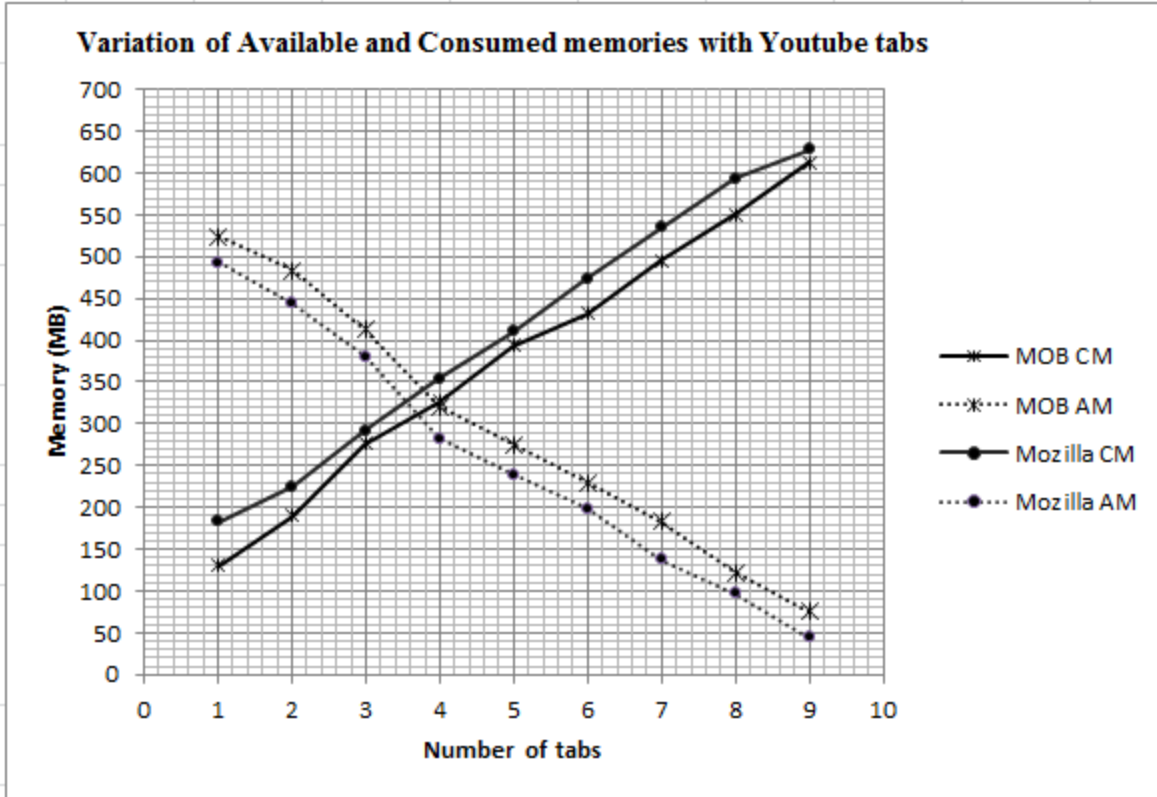


Figure 5.2: Variation of Available and Consumed memories with YouTube tabs (Source: Research)

### 5.1.3.3 Variation of Available and Consumed memories with Facebook tabs

Consumed memory increases linearly with the number of tabs. In MOB, consumed memory increases by 76.99 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $C = 76.99t + 108.65$ . In Mozilla Firefox, consumed memory increases by increases by 66.55 MB relative to 1 unit change in a browser tab. Regression model is expressed as  $C = 66.55t + 205.9$ . The value  $C$  in the equations represents consumed memory and  $t$  represents the number of tabs. In both browsers, memory consumption is directly proportional to the number of tabs with a high prediction of 98.23% and 99.34% on MOB and Mozilla Firefox respectively. The study established that there is a statistically significant association between browser tabs and consumed memory.

Available memory decreases linearly with an increase in the number of tabs. In MOB, available memory decreases by 58.45 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $A = -58.45t + 486.13$ . In Mozilla Firefox, available memory decreases by 48.62 MB relative to 1 unit change in browser tab and regression model is expressed as  $A = -48.62t + 397.21$ . The value  $A$  in the equations represents available memory and  $t$  represents the

number of tabs. In both browsers, available memory is inversely proportional to the number of tabs with a high prediction of 95.67% and 96.06% on MOB and Mozilla Firefox respectively.

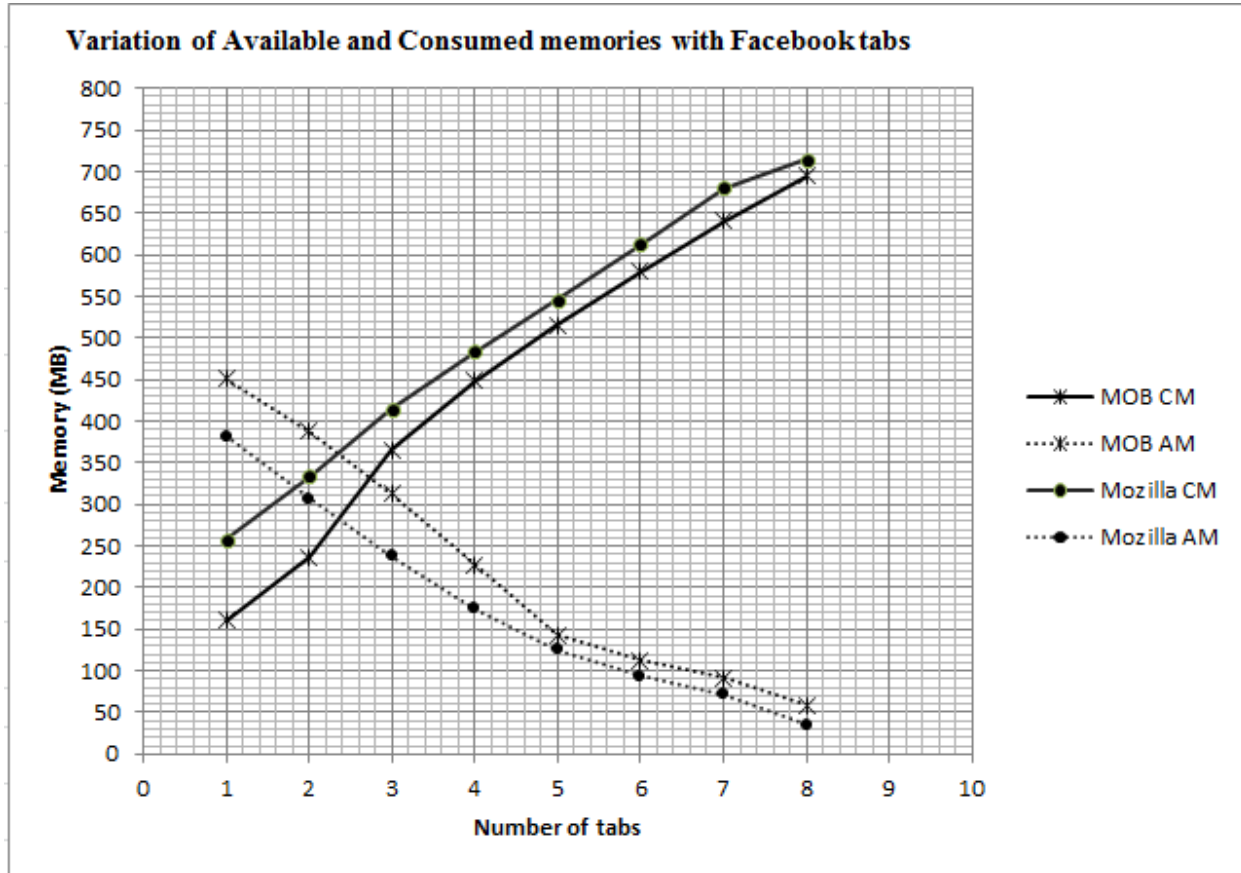


Figure 5.3: Variation of Available and Consumed memories with Facebook tabs (Source: Research)

Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is 0.229, which is greater than the chosen  $\alpha$  hence there is no statistical difference in memory consumption between MOB and Mozilla Firefox on Google. The mean difference in memory consumption is 32.76 MB

#### 5.1.3.4 Variation of Available and Consumed memories with Gmail tabs

Consumed memory increases linearly with the number of tabs. In MOB, consumed memory increases by 84.25 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $C = 84.25t + 91.789$ .

In Mozilla Firefox, consumed memory increases by increases by 83.05 MB relative to 1 unit change in the browser tab. Regression model is expressed as  $C = 83.05t + 146.47$ . The value  $C$  in the equations represents consumed memory and  $t$  represents the number of tabs. In both

browsers, memory consumption is directly proportional to the number of tabs with a high prediction of 95.35% and 96.66% on MOB and Mozilla Firefox respectively. The study established that there is a statistically significant association between browser tabs and consumed memory.

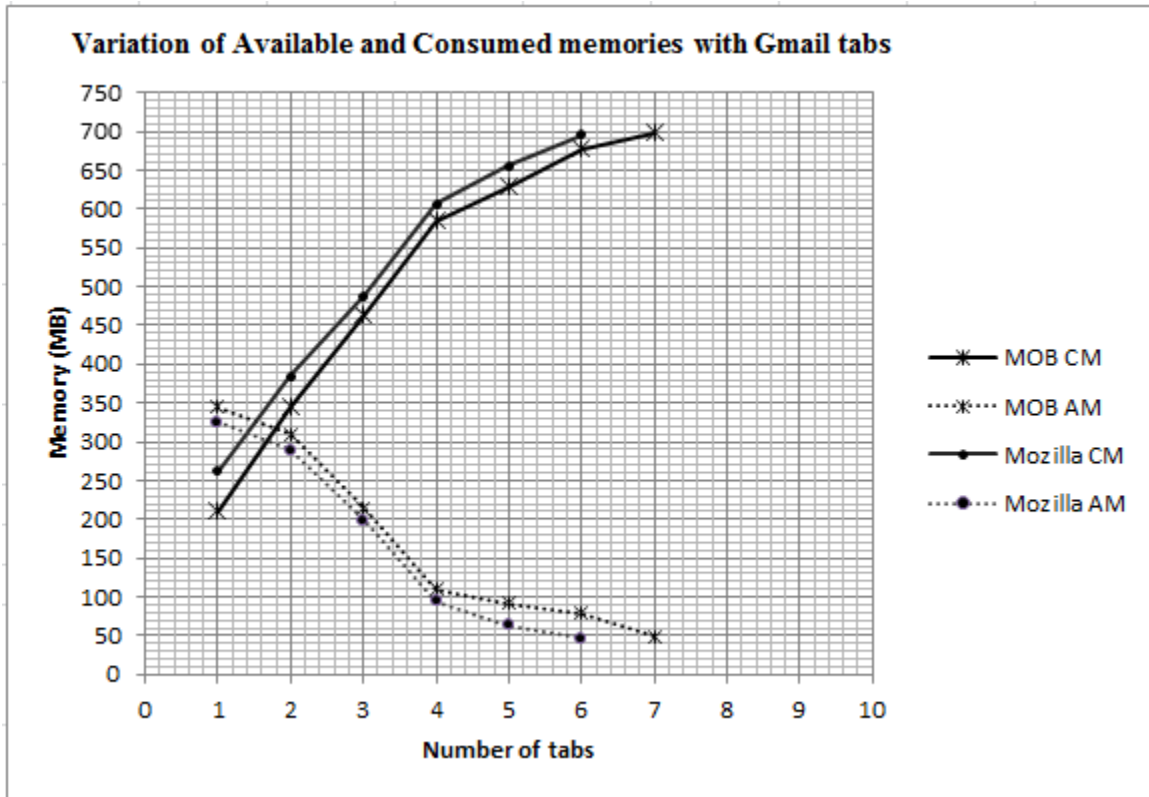


Figure 5.4: Variation of Available and Consumed memories with Gmail tabs (Source: Research)

Available memory decreases linearly with an increase in the number of tabs. In MOB, available memory decreases by 52.41 MB relative to 1 unit change in the browser tab and a regression model is expressed as  $A = -52.41t + 381.11$ . In Mozilla Firefox, available memory decreases by 62.15 MB relative to 1 unit change in the browser tab and a regression model is expressed as  $A = -62.15t + 386.84$ . The value  $A$  in the equations represents available memory and  $t$  represents the number of tabs. In both browsers, available memory is inversely proportional to the number of tabs with a high prediction of 91.29% and 94.55% on MOB and Mozilla Firefox respectively

Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is 0.29, which is greater than

the chosen  $\alpha$  hence there is no statistical difference in memory consumption between MOB and Mozilla Firefox on Google. Mean difference in memory consumption is 30.78 MB.

#### ***5.1.3.5 Deduction on consumed and available memories with homogeneous website tabs.***

The study established that MOB consumed less memory as compared to Mozilla Firefox in all tested websites. Memory consumption difference was statistically significant on Google tabs while the same was statistically insignificant on YouTube, Facebook, and Gmail tabs. Mean consumption difference for all websites combined was statistically significant with a p-value of 0.025, which is less than the chosen  $\alpha$  of 0.05. MOB consumed 38.65 MB less than Mozilla Firefox on average. MOB web browser froze computer later than Mozilla Firefox did. This was influenced by the Garbage Collector (GC), which was embedded in the memory analyzer integrated into the MOB web browser. The GC reclaimed unused memory from MOB browser objects that were unused making it consume less and ultimately gave rise to available memory.

#### **5.1.4 Memory consumption by MOB and Mozilla Firefox with heterogeneous website tabs**

Consumed memory and available memory readings were made while opening a combination of two or more tabs to depict the behavior of most web users. Analysis and discussion are done subsection herein.

##### ***5.1.4.1 Memory consumption by MOB and Mozilla Firefox for two heterogeneous website tabs***

Table 5.4 illustrates memory consumption for the two browsers by opening a combination of two various website tabs. The results in the table demonstrate that opening an additional tab raises memory demand to load the retrieved contents. The amount of memory required for the extra tab is not proportional to the memory required by opening the first tab since tabs share the memory allocated to the browser process. Browser tabs are executed as threads within the browser process. Figure 5.5 shows that a combination of Google and YouTube consumed the least memory while Gmail and Facebook consumed the highest. Computer users with the browsing habit of chatting on Facebook and reading mails from Gmail are likely to experience browser crawl much earlier than those who perform a search in the web at the same time streaming video on YouTube.

Both Facebook and Gmail comprise of sophisticated protocols characterized by background processes that execute often as users interact with the application. Executions of these processes

require high computational resources like Central Processing Unit (CPU) time and physical memory.

Table 5.4: Browser memory consumption in MB for two tabs

Website/ Browser	TEST										Avg.	S.D	
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10			
Google & YouTube	MOB	142.5	142.3	140.9	140.2	140.1	141.1	141.2	143	140.6	141.9	141.4	0.9
	Mozilla Firefox	194.4	193.9	192.6	194.1	193.5	192.3	192.7	193.8	192.9	192.3	193.3	0.7
Google & Facebook	MOB	167.4	167.3	160.6	167.9	160.9	165.8	160.9	160.2	161.7	165.3	163.8	3.0
	Mozilla Firefox	215.3	215.5	215.0	217.8	216.3	218.0	218.2	215.8	220.1	217.9	216.9	1.6
Google & Gmail	MOB	222.2	218.9	221.6	219.0	223.0	224.1	224.1	225.4	225.1	224.5	222.8	2.2
	Mozilla Firefox	260.2	264.7	267.5	268.6	268.6	268.8	268.8	266.9	267.0	267.5	266.9	2.5
Facebook & YouTube	MOB	212.6	218.9	221.6	219.0	223.0	224.1	224.1	225.4	225.1	224.5	221.8	3.8
	Mozilla Firefox	302.7	308.0	307.1	303.6	305.0	311.2	313.2	302.1	306.7	315.2	307.5	4.2
Gmail & YouTube	MOB	290.3	293.9	296.1	299.8	291.5	294.2	294.4	296.4	296.6	293.7	294.7	2.5
	Mozilla Firefox	310.4	314.7	316.3	316.6	318.8	313.8	315.2	317.9	313.3	316.6	315.4	2.3
Gmail & Facebook	MOB	312.2	309.8	312.5	311.3	310.2	312.0	313.3	312.9	311.0	313.8	311.9	1.2
	Mozilla Firefox	350.2	350.6	351	349.9	350.6	355.6	355.4	357.6	355.0	357.9	353.4	3.0

(Source: Research)

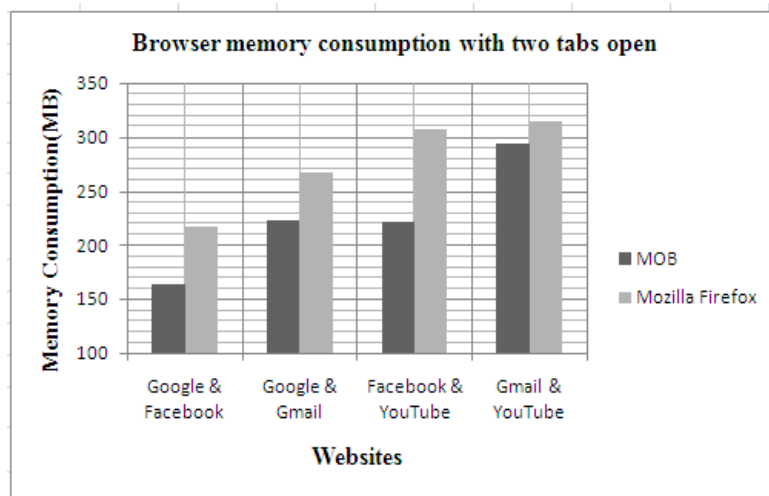


Figure 5.5: Browser memory consumption for a combination of two various websites (Source: Research)



### 5.1.4.2 Memory consumption by MOB and Mozilla Firefox for three heterogeneous website tabs

Table 5.5 illustrates memory consumptions for the two browsers by opening a combination of three various website tabs.

Table 5.5: Browser memory consumption (MB) for a combination of three various websites

Website/ Browser	TEST										AVG	S.D	
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10			
Google, Gmail & Facebook	MOB	318.2	317.8	317.5	317.3	319.6	317.1	319.4	318.2	317.3	316.5	317.9	0.9
	Mozilla Firefox	377.5	376.7	378.9	378.7	378.9	376.4	377.1	378.6	376.7	377.0	377.6	0.9
Google, Gmail & YouTube	MOB	302.4	302.2	302.4	303.1	304.1	304.9	303.5	303.5	303.8	303.6	303.4	0.8
	Mozilla Firefox	340.3	341.2	343.8	341.1	342.1	341.3	340.2	343.0	343.6	342.3	341.9	1.2
Facebook, Gmail & YouTube	MOB	362.7	363.5	361.8	364.8	367.2	367.8	363.0	366.9	361.7	368.2	364.8	2.4
	Mozilla Firefox	402.2	406.2	403.4	404.8	399.7	407.9	400.3	404.6	403.9	407.6	404.1	2.6
Facebook, Google & YouTube	MOB	218.2	218.5	219.0	219.1	219.6	218.5	218.2	218.6	218.5	218.1	218.6	0.4
	Mozilla Firefox	318.6	320.7	317.6	318.7	320.1	320.1	322.3	322.6	322.4	323.5	320.7	1.9

(Source: Research)

The results in the table demonstrate that opening an additional tab raises memory demand to load the contents. The amount of memory required for the extra tab is not proportional to the memory required by opening the first two tabs since tabs share the memory allocated to the browser process. Figure 5.6 indicates that a combination of Gmail, Facebook, and YouTube pose the highest demand for memory while a combination of Google, YouTube, and Facebook poses the least demand for memory. Computer users with browsing habits on Facebook, Gmail, and YouTube sites are likely to experience system crawl much earlier than those who stream video on YouTube, chat on Facebook and perform a web search on Google at the same time.

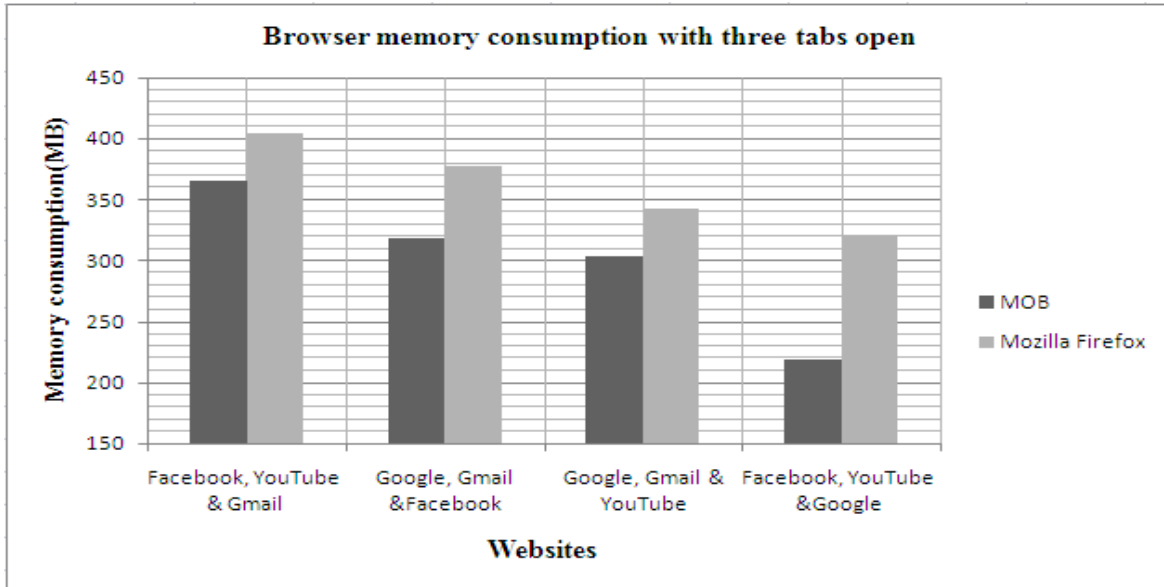


Figure 5.6: Memory consumption for a combination of three various websites (Source: Research)

#### 5.1.4.3 Memory consumption by MOB and Mozilla Firefox for four or more heterogeneous website tabs

Table 5.6 illustrates memory consumptions for the two browsers by opening a combination of four or more various website tabs. The results in the table demonstrate that opening an additional tab raises memory demand to load the contents. The amount of memory required for the extra tab is not proportional to the memory posed by opening the first three tabs and so on since tabs share the memory allocated to the browser process.

Results indicate that a combination of Google, Gmail, Facebook, and YouTube poses a memory demand of at least 447.6 MB and 479.3 MB for the MOB and Mozilla Firefox respectively. Adding more tabs raises the memory demand to a point where the computer freezes. The additional tabs from the fifth tab comprised of YouTube data. While Facebook, Gmail, and Google were possible websites to use from the fifth tab, YouTube was more popular than the latter. Memory consumption of at least 656 MB and 658.2 MB in MOB and Mozilla Firefox respectively led to computer freezing. Memory demand for MOB was considerably less than that of Mozilla Firefox in all tested cases. X represents computer freeze and (–) means no value.

Computer system froze by opening beyond seven and eight tabs in Mozilla Firefox and MOB respectively.

Table 5.6: Memory consumption (MB) for a combination of four or more various websites

Tabs/ Browser	TEST										Avg.	S.D	
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10			
4	MOB	448.8	454.6	453.2	447.6	455.7	452.3	452.6	455.7	455.8	453.3	452.9	2.7
	Mozilla Firefox	481.6	482.5	479.3	483.4	484.2	494.5	495.0	492.0	492.1	493.5	487.8	5.8
5	MOB	488.4	496.4	197.8	487.9	500.9	500.7	501.1	494.3	490.9	496.9	465.5	4.7
	Mozilla Firefox	517.4	517.5	517.3	517.1	517.4	516.9	516.7	516.6	516.7	516.6	517.0	0.3
6	MOB	542.1	548.2	535.9	553.4	545.4	551.8	548.4	552.0	541.2	545.1	546.3	5.2
	Mozilla Firefox	606.7	606.6	606.3	605.6	605.6	605.6	605.1	603.4	603.9	608.2	605.7	1.3
7	MOB	612.6	614.6	613.6	616.7	619.8	620.8	612.9	617.9	617.8	613.7	616.0	2.8
	Mozilla Firefox	658.4	661.5	656.8	653.1	650.7	651.8	663.1	663.3	668.2	655.2	658.2	5.4
8	MOB	652.6	654.6	653.6	656.7	659.8	660.8	652.9	657.9	657.8	653.7	656.0	2.8
	Mozilla Firefox	X	X	X	X	X	X	X	X	X	X	-	-
9	MOB	X	X	X	X	X	X	X	X	X	X	-	-
	Mozilla Firefox	X	X	X	X	X	X	X	X	X	X	-	-

(Source: Research)

#### 5.1.4.4 Variation of consumed and available memories with heterogeneous website tabs

Table 5.7 illustrates how consumed and available memories vary with the number of opened heterogeneous website tabs.

There is a gradual decrease in the value of available memory as the number of tabs increases as illustrated by the results in figure 5.7. Consumed memory increases gradually as the number of tabs are increased. Computer froze when available memory falls below 65 MB and 63 MB with Mozilla Firefox and MOB running respectively. X represents computer freeze.

Table 5.7: Variation of consumed and available memories with heterogeneous website tabs

Tabs	Available Memory in MB		Consumed Memory in MB	
	MOB	Mozilla Firefox	MOB	Mozilla Firefox
1	490.8	438.8	133.6	194.4
2	434.2	384.3	226.1	275.5
3	304.9	250.0	301.2	361.1
4	187.1	154.1	452.9	487.8
5	174.5	125.2	465.5	517.0
6	112.4	92.3	546.3	605.7
7	87.7	65.4	616.0	658.2
8	63.4	X	656.0	X
9	X	X	X	X

(Source: Research)

Consumed memory increases linearly as the number of tabs increase. In MOB, consumed memory increases by 75.64 MB relative to 1 unit change in a browser tab and a regression model is expressed as  $C = 75.64t + 84.29$ . In Mozilla Firefox, consumed memory increases by 78.84 MB relative to 1 unit change in a browser tab. Regression model is expressed as  $C = 78.84t + 127.4$ . The value  $C$  in the equations represents consumed memory and  $t$  represents the number of tabs. In both browsers, memory consumption is directly proportional to the number of tabs with a high prediction of 97.3% and 98.4% on MOB and Mozilla Firefox respectively. The study established that there is a statistically significant association between browser tabs and consumed memory. Available memory decreases exponentially with an increase in the number of tabs. In MOB, available memory decays at a rate of 30% per tab and regression model is expressed as  $A = 717.7e^{-0.30t}$ . In Mozilla Firefox, available memory decays at a rate of 33% per tab and regression model is expressed as  $A = 654.3e^{-0.33t}$ . The value  $A$  in the equations represents available memory and  $t$  represents the number of tabs.

In both browsers, available memory is inversely proportional to the number of tabs with high prediction of 98.7% for both the MOB and Mozilla Firefox.

Consumed memory means between MOB and Mozilla Firefox with a confidence interval of 95% indicate that the probability that the two means were not different is 0.001, which is, less than the chosen  $\alpha$  hence there is a statistical difference in memory consumption between MOB and Mozilla Firefox on heterogeneous website tabs.

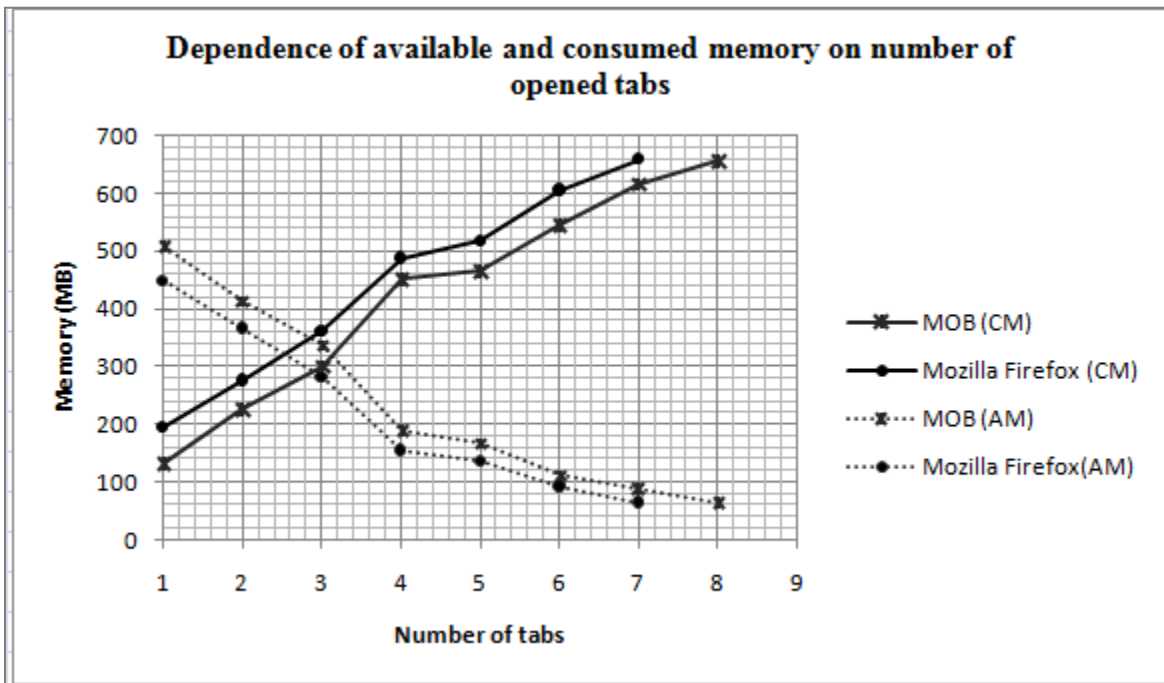


Figure 5.7: Dependence of available and consumed memory on browser tabs (Source: Research)

#### 5.1.4.5 Deduction on consumed and available memories with heterogeneous website tabs.

The study established that MOB consumed less memory as compared to Mozilla Firefox in all tested websites. On average, MOB consumed 52.08 MB less than Mozilla Firefox. The reduction in memory consumption is attributed to the enhancement made to the model, which MOB is built on. The memory analyzer was enriched with garbage collection mechanism to reclaim unused memory. The Garbage Collector freed memory thus reducing the consumed memory by reclaiming memory from browser objects whose memories were unused.

## 5.2 Hypothesis Testing

The hypothesis for this study was expressed as:

$$H_0: \mu_{\text{non-analyzer}} - \mu_{\text{analyzer}} = 0 \text{ ("the difference of the memory means is equal to zero")}$$

where  $\mu_{\text{analyzer}}$  and  $\mu_{\text{non-analyzer}}$  are the memory consumption means for browser integrated with memory analyzer and non-integrated browser respectively. With a confidence interval of 95%, the mean memory consumption for the MOB and Mozilla Firefox is 323.62 MB and 362.26 MB respectively with homogeneous tabs. The p-value is 0.025. Similarly, the mean memory consumption for the MOB and Mozilla Firefox is 289.24 MB and 341.33 MB respectively with heterogeneous tabs. The p-value is 0.001. Both cases depict lower p-value than

the chosen  $\alpha$  hence; the study established that there is a statistical difference in memory consumption between MOB and Mozilla Firefox.

### **5.3 Summary**

Memory consumption between Mozilla Firefox and MOB was compared on Google, YouTube, Facebook and Gmail websites. The comparative study aimed at finding the impact of the memory analyzer in MOB with regard to browser memory consumption. An independent t-test was adopted to deduce the difference in memory consumption was statistically significant or not. Regression analysis on obtained data aimed at deducing the rate of change of memory consumption by opening browser tabs. Conclusion about the presented results is given in chapter 6.

## CHAPTER 6

### CONCLUSION AND RECOMMENDATIONS

This chapter gives a conclusion and recommendations of the undertaken research. Section 6.1 gives a detailed conclusion of the research findings while section 6.2 enumerates the recommendations.

#### 6.1 Conclusion

This research was conducted to find out the applicability of a memory analyzer to the browser reference architecture with a view to controlling memory hogging by web browsers. A memory analyzer was developed successfully based on specified functional requirements. The analyzer was then integrated in the browser prototype and its performance on memory optimization was evaluated by comparing its memory consumption with that of the contemporary Mozilla Firefox.

The study results indicated that the integration provided a control mechanism in which the maximum amount of memory a browser would consume was set. This phenomenon, controlled browser memory hogging which consequently raised the amount of available memory.

The study confirmed that memory consumption by browsers increases by opening tabs. However, memory posed by the browser in accessing a webpage was not uniform across the investigated websites. Research findings indicate that memory consumption was dependent on the nature of web content that was fetched. Google page consumed the least memory in both browsers. YouTube page was second while Facebook was third. Gmail application was found to consume the highest memory in all tested cases.

However, memory consumption by the two browsers was different though the trend was the same. The integration of the memory analyzer in the developed browser prototype showed positive results. The study found out that the average memory consumption for memory analyzer-integrated browser was 38.65 MB and 52.08 MB less with homogeneous and heterogeneous tabs opened respectively compared to Mozilla Firefox. This was influenced by the role played by the Garbage Collector in reclaiming unused memory from browser objects.

Furthermore, it was possible to limit how much memory a browser should use with the MOB browser thus controlling memory hogging. With a memory threshold at 100 MB, a computer user would only perform a Google search. Raising TM to 200 MB allowed the computer user to perform Google search, chat on Facebook, and watch a video from YouTube with YouTube and

Facebook on one tab each. The study established that integration of the memory analyzer in the browser architectural model lowered memory consumption by the browser thus increasing the amount of available memory, which translated to improved concurrency.

The research study rejected the null hypothesis.

## **6.2 Recommendations**

The researcher recommends that further investigations to be carried out to determine the best strategy that would optimize memory as well as improve the level of concurrency in computers with less memory. Having achieved the set objectives, the research recommends the following.

- i. Further research on the performance of the developed model on computer machines having more than 1 GB RAM.
- ii. Evaluation of the developed architecture in x64 bit operating systems.
- iii. Development and integration of the memory analyzer for UNIX-based operating systems.



## REFERENCES

- Adam, O. (2011). *Web Browser Grand Prix 3: IE9 Enters The Race*. Efficiency Benchmarks: Memory Usage and Management. Retrieved from <https://www.tomshardware.com/reviews/internet-explorer-9-chrome-10-opera-11,2897.html>
- Ader, J. (2008). *Missing data*. In Ader, H. J. & Mellenbergh, G. J. (Eds). *Advising on research Methods: A consultant's companion*. (pp. 305-332). Huizen, The Netherlands: Johannes van Kessel Publishing.
- Ader, J., Gideon J. & David J. (2011). *Advising on Research Methods: A Consultant's Companion*.
- Alan, D., Barbara, H., & Roberta, R. (2012). *Systems Analysis and Design, Fifth Edition*. USA: John Wiley & Sons, Inc.
- Allan, G. & Michael, W. (2006). *Reference architecture for Web browsers.*” In: 21st IEEE International Conference on Software Maintenance (ICSM'05). pp. 661–664.
- Andre, C., Bryan, L., Neal C., Sunpreet J. & Stephen H. (2007). Conceptual Architecture of Firefox.
- Andreessen, M., Bina & Eric (1994). *"NCSA Mosaic: A Global Hypermedia System"*. *Internet Research*. Bingley, U.K.: Emerald Group Publishing Limited.
- Antero, T., Tommi, M., Dan, I. & Krzysztof, P. (2008). *Web browser as an application platform: The lively Kernel experience*. Beginners (2nd.ed.), Singapore, Pearson Education.
- Braga, M. (2011). *Web Browser Showdown: Memory Management Tested*. Retrieved from <http://www.tested.com/tech/web/2420-web-browser-showdown-memory-management-tested/>
- Brian, A., Lars B., David, H. & Josh, M. (n.d.). *Experience Report: Developing the Servo Web Browser Engine using Rust*. Mozilla research.
- Brinkmann, M. (2014). *Tools to optimize the Memory Usage of Firefox and Chrome*. Retrieved from <http://www.ghacks.net/2014/09/07/tools-to-optimize-the-memory-usage-of-firefox-and-chrome/>
- Brinkmann, M. (2018). What you can do if your browser uses too much Memory. Retrieved from <https://www.ghacks.net/2018/09/18/what-you-can-do-if-your-browser-uses-too-much-memory/>
- Chris, A. (2012). *The Man Who Makes the Future: Wired Icon Marc Andreessen*. Retrieved from [http://www.wired.com/2012/04/ff\\_andreessen/all/](http://www.wired.com/2012/04/ff_andreessen/all/)
- Chua, C.K., Leong K.F. & Lim C.S. (2010). *Rapid Prototyping. Principles and Applications*. World Scientific, New Jersey-London- Singapore-Hong Kong.
- Coates, M. (2010). *A journey in Security*. HTML5, Local Storage, and XSS. Retrieved from <http://michael-coates.blogspot.com/2010/07/html5-local-storage-and-xss.html>
- Dave, T. (2002). *"How was Mozilla born"*. Retrieved from <http://www.davetitus.com/mozilla/>
- Doug, D., (2012). HTML5 security in the modern web browser perspective. Retrieved from <https://www.nccgroup.trust/uk/our-research/html5-security-the-modern-web-browser-perspective/>

- Eldad, E. (2007). *Reversing: secrets of reverse engineering*. John Wiley & Sons.
- Fernando, O. (2013) .*Software which enables faster browsing by eliminating memory leaks in Firefox*. Retrieved from <http://firemin.en.lo4d.com/>
- Gnome desktop environment. (n.d.) Retrieved from <http://gnome.org>.
- Google developers. (n.d.). Retrieved from <https://developers.google.com/v8/intro>
- Google, (2008). Chromium blog. Retrieved from <https://blog.chromium.org/2008/09/multi-process-architecture.html>
- Gordon, W. (2017) .*Stop complaining that your browser uses lots of RAM: It is a Good Thing*. Retrieved from <https://www.howtogeek.com/334594/stop-complaining-that-your-browser-uses-lots-of-ram-its-a-good-thing/>
- Gregor, W., Andreas, G., Christian, W., Brendan, E. & Michael, F. (2011). Compartmental Memory Management in a Modern Web Browser.
- Hanson, D. R. (1990). *Fast allocation and deallocation of memory based on object lifetimes*. *Software - Practice and Experience*, 20(1):5–12, 1990. doi: 10.1002/spe.4380200104.
- Ilushin, E. & Namiot, D. (2015). JavaScript Memory Management. *International Journal of Open Information Technologies*
- Ilya, K. (2011). Memory leaks. Retrieved from <http://javascript.info/tutorial/memory-leaks>
- Jesee, B., Katricia, B., Lukas, B., Bennett, C. & Tom, F. (2009). *Conceptual architecture of Google Chrome*. ArCHROMEtecs
- Josh, M. & Keegan M. (2014, Aug 26).*JavaScript: Servo's only garbage collector*. Retrieved from <https://blog.mozilla.org/research/2014/08/26/javascript-servos-only-garbage-collector/>
- Josh, M., Brian, A. Lars, B., David, H., Keegan, M., Jack, M., Simon, S. & Manish, G. (2015, May 26). Experience Report: Developing the Servo Web Browser Engine using Rust.
- Kamau, H.,McOyowo, S. & Okoyo, H. (2018). Techniques to Control Memory Hogging by Web Browsers: An in-Depth Review. *International Journal of Computer Applications Technology and Research*, 185-192.
- Kamau, H.,McOyowo, S. & Okoyo, H. (2019). An Enhanced Browser Reference Model . *International Journal of Computer Applications Technology and Research*, 299-302.
- Karl, G. (2013).Optimize Firefox's Performance with these Memory Add-Ons! : Retrieved from <http://www.drakeintelgroup.com/2013/06/25/Firefox-memory-addons/>
- Kimak, S., Ellman, J. & Laing, C. (2014). Some Potential Issues with the Security of HTML5 IndexedDB. In: *System Safety and Cyber Security 2014 (IET Conference)*, 14-16th October 2014, The Midland Hotel, Manchester, UK.
- Klein, H. (2019). Modern Multi-Process Browser Architecture. Retrieved from <https://helgeklein.com/blog/2019/01/modern-multi-process-browser-architecture/>
- Konqueror. (n.d.) Retrieved from <http://konqueror.org>.
- Kothari, C.R. (1985). *Research Methodology- Methods and Techniques*, New Delhi, Wiley Eastern Limited

- Krill, P. (2014). *Mozilla tackles the browser memory conundrum*. Retrieved from <https://www.javaworld.com/article/2600354/java-web-development/mozilla-tackles-the-browser-memory-conundrum.html>
- Kumar, R. (2005). *Research Methodology-A Step-by-Step Guide form*. Hirzel. Connectivity-Based Garbage Collection. PhD thesis, Department of Computer Science, University of Colorado at Boulder, 2004.
- Lasar, M. (2011, OCT 11). *"Before Netscape: the forgotten Web browsers of the early 1990s"*. retrived from <https://arstechnica.com/information-technology/2011/10/before-netscape-forgotten-web-browsers-of-the-early-1990s/>
- Legan, D. (2001). *"Text-Mode Web Browsers for OS/2"*. Retrieved from <http://www.scoug.com/os24u/2001/scoug009.textbrowsers.html>
- Michael Coates, (2010). *A journey in Security*. HTML5, Local Storage, and XSS. Retrieved from <http://michael-coates.blogspot.com/2010/07/html5-local-storage-and-xss.html>
- Microsoft. (2017). *Internet Explorer Architecture*. Retrieved from <https://msdn.microsoft.com/en-us/library/aa741312%28v=vs.85%29.aspx>
- Moeskopf, E., Feenstra, F. (2008). Introduction to Rapid Prototyping. In: Raja V., Fernandes K. (eds) *Reverse Engineering*. Springer Series in Advanced Manufacturing. Springer, London
- Mozilla Developer Network. (2018). Retrieved from <https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Spidermonkey>
- Mozilla Developer Network. (2019, March 23). Common causes of memory leaks in extensions. Retrieved from [https://developer.mozilla.org/en-US/docs/Extensions/Common\\_causes\\_of\\_memory\\_leaks\\_in\\_extensions](https://developer.mozilla.org/en-US/docs/Extensions/Common_causes_of_memory_leaks_in_extensions)
- Mozilla Foundation. (2017). Module Owners. Retrieved from <http://www.mozilla.org/owners.html>.
- Mozilla Foundation. (2017). Module Owners. Retrieved from <http://www.mozilla.org/owners.html>.
- Nield, D.(2018). 9 Common Browser Problems and How to Fix Them. Retrieved from <https://gizmodo.com/9-common-browser-problems-and-how-to-fix-them-1827014349>
- Nyce, J. M. & Kahn P. (1991). *From Memex To Hypertext: Vannevar Bush and the Mind's Machine*. Academia press, San Diego.
- Otto, K. & Antonsson, E. (1991). Trade-off strategies in Enginerring Design. *Research in Engineering Design* Volume 3, Number 2 (1991), pages 87-104
- Paul, K. (2016). *Mozilla tackles the browser memory conundrum*. Retrieved from <https://www.javaworld.com/article/2600354/java-web-development/mozilla-tackles-the-browser-memory-conundrum.html>
- Paulina, S., Raúl M., & Eduardo, B. (2016). *A Reference Architecture for web browsers: Part I, A pattern for Web Browser Communication*
- Pryden, D. (2015, September 6). Why do languages such as C and C++ not have garbage collection, while Java does? [closed]. Retrieved from

- <https://softwareengineering.stackexchange.com/questions/113177/why-do-languages-such-as-c-and-c-not-have-garbage-collection-while-java-does>
- Raúl R. (2015). *Browser comparison, 2015 edition*. Retrieved from <http://blog.en.uptodown.com/browser-comparison-2015/>
- Raúl, R. (2015). *Browser comparison, 2015 edition*. Retrieved from <http://blog.en.uptodown.com/browser-comparison-2015/>
- Sagar, A., Pratik, G., Rajwin, P. & Aditya, G. (2010). *Market research on web browsers*. Retrieved from [http://www.slideshare.net/sagar\\_agrawal/research-on-web-browsers](http://www.slideshare.net/sagar_agrawal/research-on-web-browsers).
- Sebrechts, J. (2012). Why do browsers leak memory? Software Engineering. Retrieved from <https://softwareengineering.stackexchange.com/questions/173627/why-do-browsers-leak-memory>
- Serea, R. (2019). SpeedyFox 2.0.26 Build 140. Retrieved from <https://www.neowin.net/news/speedyfox-2026-build-140/>
- SimilarWeb. (2018). *Website ranking*. Retrieved from <https://www.similarweb.com/global>
- Statista. (2018). *Operating Systems - Statistics & Facts*. Retrieved from <https://www.statista.com/>
- Tali, G. & Paul, I. (2011). *How Browsers Work: Behind the scenes of modern web browsers*. Retrieved from <https://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>
- Tedd, H. (1965). *Complex information processing: a file structure for the complex, the changing and the indeterminate*. Proceeding ACM '65 Proceedings of the 1965 20th national conference pp.84-100
- Tim, B. (1999). *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco.
- Vladan, D. & Ashvin G. (2009). Securing script-based extensibility in web browsers.
- Vrbanec, T., Kiric, N. & Varga, M. (2013). "The evolution of web browser architecture". SCIECONF 2013, pp. 472–480.
- W3C. (2004). *Architecture of the World Wide Web, Volume One*. Online. Retrieved from <http://www.w3.org/TR/webarch/>
- W3C. (2017). *Browser & Platform Market Share*. Retrieved from <https://www.w3counter.com/globalstats.php?year=2017&month=12>
- Wayne, W. (2018). How to stop browser freezing. Retrieved from <https://www.cloudpro.co.uk/leadership/cloud-essentials/7443/how-to-stop-your-browser-freezing>
- William, M. (2017). About: addons-memory 12. Retrieved from <https://www.techworld.com/download/internet-tools/aboutaddons-memory-12-3329018>
- Xi, Y., Zhichao, H., Bin, H., Senjie, Z., Chao, W., Feifei, G., & Shi, J. (2017). *RaPro: A Novel 5G Rapid Prototyping System Architecture*. IEEE Wireless Communication Letters.

# APPENDICES

## Appendix I: Research permit.



**NATIONAL COMMISSION FOR SCIENCE,  
TECHNOLOGY AND INNOVATION**

Telephone: +254-20-2233871  
2241349, 2310971, 2219420  
Fax: +254-20-116245, 118249  
Email: [info@nacosti.go.ke](mailto:info@nacosti.go.ke)  
Website: [www.nacosti.go.ke](http://www.nacosti.go.ke)  
When calling please dial

NACOSTI, Upper Kabete  
Off. Thurochi Way  
P.O. Box 29673-00100  
Nairobi-KENYA

Ref. No. **NACOSTI/P/19/15581/29742** Date **23<sup>rd</sup> May, 2019**

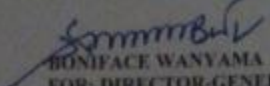
Harun Kariuki Kamau  
Maseno University  
Private Bag  
**MASENO.**

**RE: RESEARCH AUTHORIZATION**

Following your application for authority to carry out research on *"Integration of the memory analyzer to the browser reference architecture"* I am pleased to inform you that you have been authorized to undertake research in **Kisumu County** for the period ending **23<sup>rd</sup> May, 2020**.

You are advised to report to the **County Commissioner, and the County Director of Education, Kisumu County** before embarking on the research project.

Kindly note that, as an applicant who has been licensed under the Science, Technology and Innovation Act, 2013 to conduct research in Kenya, you shall deposit a **copy** of the final research report to the Commission within **one year** of completion. The soft copy of the same should be submitted through the Online Research Information System.

  
**BONIFACE WANYAMA**  
**FOR: DIRECTOR-GENERAL/CEO**

Copy to:

The County Commissioner  
Kisumu County.

The County Director of Education  
Kisumu County.