Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Performance Analysis of JavaScript

by

Fredrik Smedberg

LIU-IDA/ LITH-EX-A--10/020-SE

2010-05-17

Linköpings universitet

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Performance Analysis of JavaScript

by

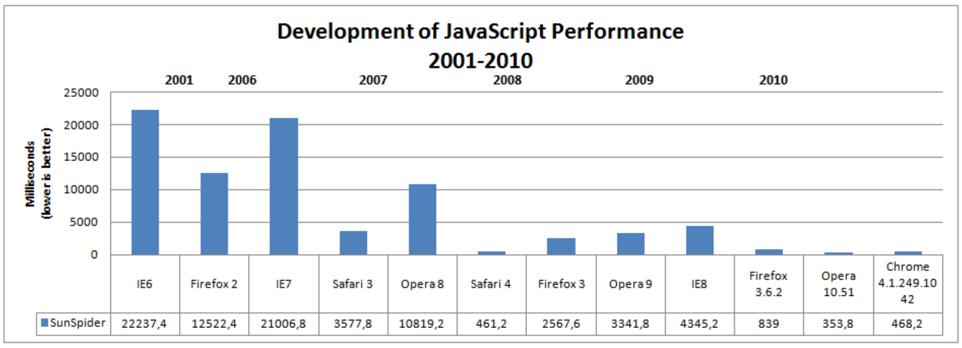
Fredrik Smedberg

LIU-IDA/LITH-EX-A--10/020-SE

2010-05-17

Supervisor: David Byers
Examiner: Nahid Shahmehri

Summary



	IE6	Firefox 2	IE7	Safari 3	Opera 8	Safari 4	Firefox 3	Opera 9	IE8	Firefox 3.6.2	Opera 10.51	Chrome 4.1.249.10 42
■ SunSpider	22237,4	12522,4	21006,8	3577,8	10819,2	461,2	2567,6	3341,8	4345,2	839	353,8	468,2

Figure 1: Development of JavaScript Performance over Time

In the last decade, web browsers have seen a remarkable increase of performance, especially in the JavaScript engines. JavaScript has over the years gone from being a slow and rather limited language, to today have become feature-rich and fast. It's speed can be around the same or half of comparable code written in C++, but this speed is directly dependent on the choice of the web browser, and the best performance is seen in browsers using JIT compilation techniques.

Even though the language has seen a dramatic increase in performance, there's still major problems regarding memory usage. JavaScript applications typically consume 3-4 times more memory than similar applications written in C++. Many browser vendors, like Opera Software, acknowledge this and are currently trying to optimize their memory usage. This issue is hopefully non-existent within a near future.

Because the majority of scientific papers written about JavaScript only compare performance using the industry benchmarks SunSpider and V8, this thesis have chosen to widen the scope. The benchmarks really give no information about how JavaScript stands in comparison to C#, C++ and other popular languages. To be able to compare that, I've implemented a GIF decoder, an XML parser and various elementary tests in both JavaScript and C++ to compare how far apart the languages are in terms of speed, memory usage and responsiveness.

Thanks to

Thanks to Opera Software and Linköping University that made this thesis possible. Special thanks to Nicklas Larsson (my supervisor at Opera), Geoffrey Sneddon, James Graham, Daniel Spång, Jens Lindström and the other people at the Core division of Opera Software for answering all my silly questions.

I also want to thank Professor Nahid Shahmehri and David Byers at the ADIT division of the Department of Computer and Information Science at Linköping University for their guidance while writing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 A Short History of the Web

Before the Internet became a part of every user's daily life, software was developed and distributed on floppies, CDs and using phone lines calling bulletin board systems. Distribution was cumbersome, time consuming and expensive. If bugs were discovered, it often took weeks or months before they were reported, fixed and an updated version distributed. Viruses spread at a slow rate, often from computer to computer by floppies. Spam was unheard of.

When users began adopting the Internet everything changed. To name a few examples, users started browsing electronic pages using a web browser, began e-mailing each other, started using instant messaging software like ICQ and MSN, began receiving lots of spam advertises and started making use of mailing lists to solve problems together. Faster communication revolutionized the progress of development. Before the Internet, it made no sense to outsource part of the production to a country like Sri Lanka. The communication costs would have made it too expensive, if the parties even discovered each other. The Internet has made it possible for almost everyone to communicate and learn, regardless of distance or financial status [1].

The introduction of the browser, and the web wars between Microsoft and Netscape that followed, speed up the invention of new features available to web developers. To make web pages more interactive and less static, Netscape invented JavaScript. Microsoft released their own implementation called JScript. Both vendors had their own view of what their browsers should be able to handle and no real standard was established. Private companies and governments moved out to the web by introducing online banking, e-government services and e-commerce. In the years that followed, the web began to slowly become standardized with some vendors accepting the standards more readily than others, and new web browsers were born. The Mozilla Foundation released Firefox, Opera Software introduced Opera, Apple released Safari and later on Google gave birth to Chrome.

Over the years, web pages have become increasingly interactive and social, adopting the usability and visual form of regular desktop applications. This change is sometimes called glocalization. The term means someone that can sit at any desktop around the world, using the same files and applications she would be using at home, while for example, being able to communicate with her colleagues at the other side of the globe. The role of the single computer is marginalized.

To develop web applications that behave and have the same functionality regardless of which web browser the end user is using, standards or widely accepted technology needs to exist. Flash and Java have together with standards for HTML, XHTML, CSS and JavaScript dominated the web. Far from every browser vendor fully supports these, and Microsoft with their Internet Explorer is often considered the worst. However, the trend is towards standard compliance and even Microsoft tries to adopt to this [2].

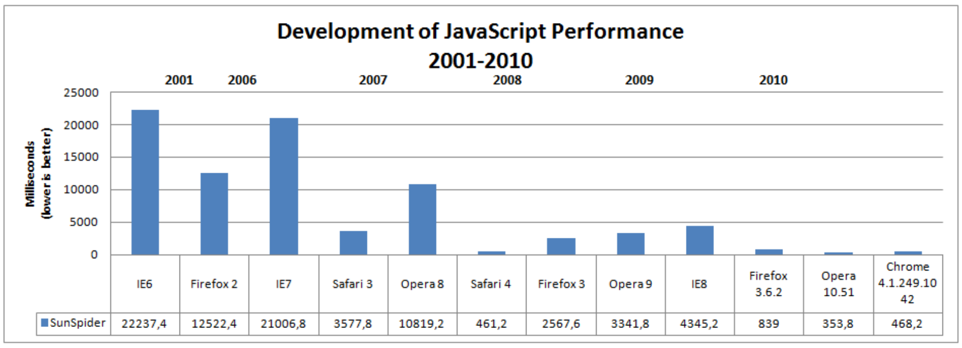
1.2 The Need for Increased Speed

When Microsoft and Mozilla added XMLHttpRequest to JavaScript, and Jesse James Garrett introduced the idea of AJAX technology in 2005, the popularity of JavaScript grew rapidly [3]. Google pioneered this field by using AJAX in their new Gmail and Maps services [4]. Other major players followed and soon browser vendors realized that a growing bottleneck was the performance of JavaScript. Between 2007 and 2008, JavaScript engines underwent a tremendous boost in speed. Some were redesigned, others optimized. In most implementations, the speed of JavaScript was increased by 2-3 times! These figures will later be confirmed by several tests performed in this thesis.

A new generation of JavaScript engines followed, which made use of *just in time* compilation, trace trees, better garbage collecting and more effective code parsing. These new innovations opened up doors for creative developers to create more resource hungry and complex software. The kind of applications that before this technology saw the light of day, had required compiled languages like C++, to run satisfactorily. This move meant a lot for end users aiming to become glocal. The speed increase did not only make browsing faster; it also boosted creativity and developers created implementations of JPEG encoders [5], Nintendo emulators [6] and full desktop-like web sites. That was simply not feasible some years ago.

The need for increased speed is not just a matter of increased application throughput. Sometimes, it's more important to be able to plan, develop and deliver a product on short notice, instead of focusing on execution speed. Often, less complexity is favored while accepting a somewhat lower performance. For this, languages like Ruby, Python, JavaScript and Java exist.

After all, the transistors in computers have almost doubled every one and a half year since the introduction of Moore's law [7], resulting in increased performance.



	IE6	Firefox 2	IE7	Safari 3	Opera 8	Safari 4	Firefox 3	Opera 9	IE8	Firefox 3.6.2	Opera 10.51	Chrome 4.1.249.1042
■ SunSpider	22237,4	12522,4	21006,8	3577,8	10819,2	461,2	2567,6	3341,8	4345,2	839	353,8	468,2

Figure 1.1: Performance Time Line of JavaScript

1.3 Purpose and Problem Formulation

In large software projects with a large number of developers, it's important that source code is predictable, stable, easy to understand and causes few security issues. To create a product with good performance, low memory consumption and with few bugs isn't trivial. The quality of a final version of a software product can vary greatly depending on the developers experience, the work hours spent, project management, chosen software language etc. One purpose of this thesis was to identify some of those parameters that determine the development of software. Another was to challenge the prejudice that a dynamic high level language can't be equally fast as a traditional compiled one, like C++.

This thesis was written while working at Opera Software. The company has a fast JavaScript engine, but a new generation of this engine has been created that will boost the execution speed noticeably. This increase in speed creates new usage possibilities, making it feasible to use JavaScript productively and effectively in areas where it hasn't been considered before. This thesis will try to prove that many old aspects in terms of usage no longer hold.

Most of Opera's products are written in C++, Pike and dialects of Java. One of them, their web browser core, was modified to see if certain JavaScript functions benefits from being implemented directly in the engine. That, and some other external software, have formed the basis for the hypothesis that a great amount of code is unnecessarily written in C++, and would benefit from being written in a high level language like JavaScript, letting the developer avoid the allocation of memory, making mistakes with pointers, making wrong casts etc.

It's naturally not given that rewriting code will mean an improvement. In fact, it might be easier to read and understand, but with the side effect of introducing new bottlenecks, such as insufficient development utilities. With sub optimal tools, the time needed to complete tasks might increase greatly, even though the language should be easier to use [8]. To find an optimal measurement for when it's suitable to rewrite code is near to impossible. The number of successful hacking attempts, working time spent, the number of crashes and bugs reported, are examples of questions that can't be answered satisfactorily after only six month of examination.

The results in this thesis have been collected to point out how suitable it is to use JavaScript for heavier application-like web sites, rather than only for creating superficial web design effects.

A growing trend among software companies is to gradually abandon traditional software engineering methods and instead move to a web based application approach, often called *web 2.0* [9]. This model is especially attractive from the aspect that errors can be corrected in one place instead of having to send out software updates to every client running the product. Because of this, developers can put their focus on functionality rather than on specific solutions for a particular computer or architecture, as JavaScript is entirely hardware abstract. Because web applications only require an installed web browser and that all saved data in addition is centralized, it can make backups easier, reduce computer specific risks and probably reduce a company's costs to maintain clients significantly. Of course, there are risk of increased side effects such as lower performance, higher memory usage, lower responsiveness and that the developers' experience less satisfaction with their change of development tools.

1.3.1 Questions

To say whether a dynamic language can perform somewhat on par with C++, the following questions have been investigated in C++ code rewritten in JavaScript.

- Is the performance of the software affected?
- Has the code become easier to comprehend?
  - How many lines of code is the result?
  - Is it reusable?
  - Does it introduce memory leaks?
- What development utilities exist and what's their impact on the development?
  - In Windows?
  - In Linux?
- When and where is it motivated to use JavaScript?
- How often is the code used, every millisecond, minute, hour or day?

1.3.2 The Chosen Languages

When choosing programming languages, I wanted a popular traditional compiled language and a less complex but widespread one. C++ was chosen as the traditional language, because it's multi-paradigm, statically typed, compiled, considered very fast and commonly used among people developing high end server software, computer games and multimedia codecs. The language with less complexity would ideally have characteristics of the traditional one, making it easier to port test software and being familiar to the readers of this paper.

JavaScript was given the role of representing the high level family, as there are many different implementations with a wide variation in performance and memory usage. It's available on a variety of platforms like PC, Mac, handheld computers, phones, Nintendo Wii and PlayStation 3. By design, it's extremely dynamic, prototype based, dynamically typed and widely used across the web on clients and some servers, and also part of popular software like Adobe Acrobat and Microsoft's .NET Framework, to name a few. The syntax of the language is similar to C++, easing the task of porting and comparing software between the two languages.

Python, Ruby and Java were not considered. I needed a language affecting end users on a daily basis, where the software runs on their individual machine, with performance and memory usage affected by the end users' choice of software, i.e. the web browser. Python is often used as a script language on servers, and to dynamically serve web pages, but the end user can seldom change the performance or behavior by making changes to their own desktop. Ruby was dropped of the same reason. Applications written in Java are found on both desktops and servers, ranging from end user software to high end banking applications. However, there exist few, really only one, reasonable virtual machine, making it hard for any end user to optimize memory usage or performance, without upgrading their hardware.

1.4 Outline of this Thesis

The organization of this thesis is as follows: it begins with an *Introduction* followed by a chapter giving *A Short Introduction to JavaScript*, explaining what version has been used and the basics of a virtual machine. It's followed by a thorough description of the *Areas of Interest*, explaining what will be investigated and what won't. The *Related Work* chapter then comments on some scientific papers that have researched the performance and memory usage of JavaScript in the past. The *Methodology* then explains the steps taken to produce the necessary results to evaluate the areas of interest.

After the methodology, the *Implementation* of the software that were made to collect the results are explained, beginning with their individual purpose, then the design and last the implementation. The implementation is followed by *Results* chapter that presents all the findings of the tests run. First some detailed and more interesting results, and in the end more generic results from industry benchmarks. After the results have been presented, the thesis ends with a *Discussion*, *Conclusion* and *Future Work* chapter.

A *Glossary* can be found at the end of the thesis, before the *Bibliography.*
N.B. JavaScript and JS are used interchangeably within the thesis.

Chapter 2

A Short Introduction to JavaScript

2.1 The Birth

In 1995, when the web was relatively young and the majority of the users were using a web browser from Netscape, no browser had support for any kind of scripting or programming. Brendan Eich, working at Netscape at the time, was asked to invent a language that could make the web more dynamic and interactive. To maximize adoption among developers and amateurs, the language was designed to be very easy to learn. At first, it was named Mocha, later LiveScript and by the end of 1995, Netscape and Sun did a license agreement and changed the name to JavaScript.

People quickly adopted the language. Given its growing popularity, Microsoft was forced to implement their own compatible dialect called JScript. In 1996, Netscape submitted JavaScript for standardization by Ecma International, whom released a first edition of the standard in the middle of 1997, ECMA-262. ECMA-262, edition 3, is the JavaScript standard that's supported by leading web browsers today [10].

2.2 Language Design

ECMA Script is an object oriented prototype based language with a similar syntax to C and Java. The prototype based design facilitates easy delegation and flexible overriding of object behavior. There's no need to specify data types when defining variables or to allocate memory. It's handled automatically by the virtual machine.

Below is an example of how prototype based inheritance works. More examples of what ECMA Script looks like can be found on ecmascript.org.

Listing 2.1: Sample code

```
// This is the base object.
function Car() { }
Car.prototype = new Object();
Car.prototype.wheels = 4;
Car.prototype.color = "black";

// This is an object that inherits the car objects
// prototype-data, but overrides the color.
function RaceCar() { }
RaceCar.prototype = new Car();
RaceCar.prototype.color = "red";

var vroom = new RaceCar();
vroom.wheels // 4
vroom.color  // "red"
```

2.3 The Virtual Machine

A virtual machine creates a virtual environment for a language or an architecture. Some virtual machines are used to emulate entire systems while others creates a running environment for a specific language. When imagining a modern virtual machine made to run JavaScript, one can picture a dynamic compiler but without some of the traditional compilation steps.

The VM parses the JavaScript and creates a parse tree which it converts to byte code. While going through all this byte code, it keeps track of code blocks that are run several times, and would benefit from being converted to real machine code. The byte code that isn't transformed is made efficient by attaching very fast paths to the needed system call functions.

Creating an intelligent and fast parser is half of the equation. The other half is having a memory efficient garbage collector. Most users have multiple web sites open, sites which normally use lots of JavaScript. They easily consume up to 20 MB of memory each, depending on the complexity and characteristics of the running JavaScript application [11]. When the user navigates from page to page, it's a requirement to keep track of what memory can be freed, as otherwise the system might run out of memory.

This thesis won't do any deep analysis of modern JavaScript VMs because VM design can easily become a thesis itself. The only analysis that will be made are superficial measurements of memory consumption, performance and response times.

2.4 Strategies Towards Speed

The first generation of JavaScript engines were pure parsing engines. They went through the script, created a parse tree and executed the instructions, often consuming a lot of memory. The second generation optimized this process by converting the parse trees to byte-code. However, the real speed boost came with the introduction of JIT[1] compilation. It means converting code at run-time prior to executing it natively, by first analyzing and optimizing the byte-code, then converting parts of the code, or all, to machine code. This technique has increased the speed of execution by a magnitude at the trade-off of increasing the start-up time, and is the technology currently used in the fastest JavaScript engines.

[1]Just-in-time

Chapter 3

Areas of Interest

3.1 How Performance gets Affected

The focus on performance is limited to

- The time needed for the code to perform a given task
- The maximum amount of memory allocated during a run
- Responsiveness in the form of how quickly the software start and begin performing its task
- The possibility of parallel execution

And will be examined by

- Rewriting JavaScript to C++. The hypothesis is that the correctly chosen code, rewritten to C++, can impact the whole performance of a JS product dramatically.
- Rewriting C++ to JavaScript. The hypothesis is that large amounts of source code is unnecessarily written in C++ and would run almost as fast in JS in a well written VM.

When measuring performance, a timer was placed measuring the time over the exact code block, ignoring the start-up time of the program and code run before or after the investigated block. The measurements were run while no other software was running, except that needed by the operating system. The average speed was calculated by running ten iterations of the investigated code block.

Different kinds of applications were implemented to measure the development time, the time taken to debug and solve errors, and the design of the architecture [12].

Responsiveness was measured by looking at how quickly the software loaded and started to perform its task.

The tests weren't planned to implement the use of threads to test parallel execution. When this thesis was first written and the software implemented, C++ allowed threading while JavaScript didn't. At the end of writing the thesis, many JS engines had added support for threads as specified in the WebWorkers draft published by WHATWG[1] [13]. To take advantage of this, some tests were run using a threaded JavaScript JPEG encoder [5].

3.2 The Amount of Code and its Influence on Understandability

The change in code size, was measured by counting how many lines of code that were needed to do the same task in the compared languages. Because code can be written to an unreadable minimum, a core requirement was the use of common sense. That means to not write unreadable one liners.

Fewer lines of code were expected to make the reading easier. A concrete example of this was searching for data in strings. While a simple search may only require one line of code in JS, the same task done in C++, may require a lot more. This assumes the programmer doesn't use the String library available in C++'s Standard Template Library. That assumption is fair, considering many companies, like Opera, implement their own String libraries to decrease binary size and facilitate portability.

3.3 Memory Leaks

While allocation of memory is handled automatically by the JavaScript-engine, badly written code can result in unnecessary large allocations. A famous example is the usage of circular references, which can leak a lot of memory in some JS and DOM implementations [14]. However, most modern engines implement methods to avoid these kinds of worst case scenarios.

The software used in this thesis don't try to deliberately create memory leaks.

3.4 Portability and Reusability

Depending on the purpose of an application and its tasks, if the system is a phone, handheld or normal computer, runs on a 32 or 64 bit system, the C++ code may have to be rewritten between the platforms to give the same expected and desired behavior. JS however, is designed to be portable and easy to reuse, to behave exactly the same regardless of the system it operates on, without change to the code.

The thesis briefly evaluates the portability of the software implemented.

[1]Web Hypertext Application Technology Working Group.

3.5 Development Tools

It's natural that a language like C++, which has been around for a long time, has a wider range of development tools compared to those available for JavaScript. However, quantity don't necessarily mean something positive. The tools might be needed because the design of the language makes it hard to work without them, making the same tools for a newer and less complex language obsolete. The focus has been on finding out if available development tools create restrictions, limitations or introduce new bottlenecks. The development tools that have been examined are code editors, debuggers, profilers and compilers.

The platforms that were tested were Windows and Ubuntu Linux. The evaluation paid special attention to how the tools eased coding and compilation, and made debugging and writing correct code easier. For example, one tool allowed compilation and debugging to be integrated in the IDE, while another only functioned as an editor forcing the developer to compile and debug in the web browser or in the console. The evaluation was very subjective and I compared how I experienced their effect on my work-flow.

3.6 When, Where and how Often

A lot of code might be written in an unnecessarily powerful language. For example, video decoders are mostly written in C/C++ with additional assembler to speed up the decoding. The end-user applications that display these videos, like VideoLAN, don't have to be written in C/C++, even though they mostly are. They could primarily be written in a language like Python or JavaScript and use a highly optimized C/C++ library to decode the video.

Part of the purpose of the implementations made in this thesis have been to show how good JavaScript is at running CPU intensive tasks, to see if the majority of the applications written today could be converted to JavaScript and moved out on to the web without loosing performance.

Sometimes performance might be lost. Depending on how often a particular function are being used, it might not be an issue. A JS function that's 40% slower than the C++ equivalent, that's used only every minute, might not be a problem at all. A function that's 5% slower and used every millisecond, could, on the other hand, noticeably affect performance.

3.7 Software Language

C++ and JavaScript are the only languages that will be compared.

Chapter 4

Related Work

4.1 Comments on Sun Microsystems Research

In 2007, the scientists Mikkonen and Taivalsaari wrote a paper called *Using JavaScript as a Real Programming Language* [15], in which they sum up their experience of using it to create a web based desktop-like system called *Lively Kernel.* Consideration has to be taken to the conclusions of Mikkonen and Taivalsaari, considering the paper was written in 2007, and that the performance of JS engines have since doubled or tripled. The reader shall also note that employees of Sun Microsystems risk being subjective because of the company's historically strong preference for the Java language.

Five years ago JS was mostly used to create small effects on web pages. The language has since become standardized and received a steadily growing attention from web developers. Since the creation of the paper, several desktop-like web implementations have been published. Examples are AtomicOS and JS/UIX, both written in JavaScript. Another is MyGoya, written in Flash.

Sun used JavaScript to write Lively Kernel with the primary purpose of finding out the limits of the language, using an experimental evolutionary approach to evaluate it's suitability to write powerful software. The scientists concluded that it without a doubt was possible, however, they identified *three major concerns* that might affect development.

4.1.1 Extremely Tolerant

JS only report errors if it's absolutely necessary. For example, the data type is not verified, the specification doesn't require to look out for undefined variables or if a function returns a value. The authors of the report view this functionality in JavaScript as an issue and the tolerance for errors seems to have caused a lot of difficulties for them.

4.1.2 Lack of Modularity and Functionality

Mikkonen and Taivalsaari criticize JS deficiencies in its modularity, that it lacks the ability to include source code from other files, and has no *real* classes. The authors continue, criticizing the absence of native support for networking, graphics and media. They feel that it's something that JavaScript needs to support in order to be a complete language.

4.1.3 Memory Management

When the report was written the researchers found that JS engines often have poor memory management. It's an interesting statement. If the current generation of JS engines have become faster, then, perhaps their memory management have also seen improvements?

4.2 Comments on two Research Papers from Microsoft

JSMeter: Characterizing Real-World Behavior of JavaScript Programs och *JSMeter: Measuring JavaScript Behavior in the Wild* are the two newest reports of those I've gone through, released in December 2009 and in January 2010 [11] [16]. Unlike many other reports, the researcher Ratanaworabhan claim to have have investigated both synthetic tests and real web applications. To investigate more than benchmarks are a bit unusual, which makes these reports especially interesting.

They have done their tests by focusing on three areas, "Functions and Code", "Heap-allocated Objects and Data" and "Events and Handlers." As aid, they used a modified version of Internet Explorer's JavaScript engine with built-in support for trace output. One thing you must keep in mind when reading these reports are their relevance to this thesis. They claim to only examine the typical JavaScript performance, but in fact examine much more than that. Events are for example something very specific to DOM, where not only the performance of JavaScript is crucial.

Their thesis is that benchmarks are not representative of the types of web pages users usually visit, which may be true, but they only mention SunSpider and V8. Other tests like Peace Keeper is not mentioned at all, even though the event tests that's carried out are very DOM specific. As basis for their argumentation is the argument that those tests, in SunSpider and V8, are very compute intensive and repetitive. That they perform the same task over and over again, while web pages typically have their JavaScript emphasis in the management of thousands of events over a short time, because users usually spend a short time on each page. The average time spent on a web page typically consists of easy calculation operations, where the emphasis is more event intensive. Since no test case focuses on the engine's ability to quickly manage events, they argue that many key areas in need of optimizations might be forgotten.

Also, the authors found that the way the benchmarks allocate heap data differs markedly from how a typical web pages works. Strings and functions dominate the actual object allocation, but despite this, objects and arrays usually have a more long lived life in the engines than strings. The writers believe that this indicates that the *collection algorithms* that assumes that objects die quickly, are inefficient. The problem is that report presents very much but only in a one-sided way, focusing only on IE8. They argue that the same tests can be performed in other browsers, and that's reasonable, but without actually doing it, they can't conclude that the benchmarks used in industry is substandard. In one of the two reports, a small test was performed on Chrome, but it was so small that it didn't add anything. Significantly more extensive tests would have been needed because something like memory allocation may differ dramatically between a good and bad implementation of a dynamic language, whether it is an engine that handles Python or JavaScript.

The authors say that it's historically been proven that ineffective performance benchmarks are holding back the development of programming languages. As example, SPECjvm98[1] was long used in the Java community to evaluate Java implementations, although it was common knowledge that the test suite not focused on the right kind of problems. To remedy this, DaCapo[2], a new efficient test suite with test cases where real applications, like Eclipse, was used, was invented.

Researchers from Microsoft have a theory about what code that can benefit from JIT: so called hot functions, functions that are used frequently and repeatedly. It is surprising that Microsoft with this realization has not yet adopted some form of JIT in its JavaScript engine. What is even more surprising is that they take up a specific function such hot function from the Economist, as the terms of an "anomaly". The function consumes very large amount of byte-code if it runs in IE. The code that IE needs to run are an extra if statement, which makes the code 50% slower because of the added if statement, as the comparison is rarely likely to be false. My point is that the authors completely one-sidedly focus on performance issues in IE, which of course is positive from one aspect, but that doesn't mean that their results can be said to demonstrate that industrial benchmarks are inadequate. They mention other browsers, but presents no figures for how they stand up in the tests they have performed on IE, which makes the results difficult to evaluate.

Opera's Carakan engine behaves much like what are being described. Heuristics keeps track of which code blocks are used repeatedly and over a

[1]A Java benchmark suite used back in 1998.
[2]A Java benchmark that replaced SPECjvm98.

certain threshold JIT them. Chrome's V8 engine behave similar, but slightly more sophisticated methods are used to judge when the code needs to be JITed.

The report argues that the typical web pages can have up to 2 MB of JavaScript code, but usually only use about 50-200 KB of this source. This makes sense since the web pages that use JavaScript frameworks like Scriptalicious, jQuery and others often use only a fraction of their functionality. Another interesting finding, which I can confirm with my discoveries, is that JavaScript programs can quickly draw a lot of memory. A web page like Bingmap, Facebook, Gmail and others can use up to 20 MB in short intervals, the report says. It must be remembered that these measurements, although they claimed to be made solely on jscript.dll, probably containing a mixture of bitmap data from the DOM tree, the JavaScript data, etc. Bingmap and Facebook are very image intensive, and if the DOM domain in these pages interact intensively with the JavaScript domain, memory consumption will grow quickly.

What I found most interesting in the report were the new issues they raise, the importance of that the JavaScript engine not only needs to be fast in benchmarks, but also its impact on the responsiveness, effective management of events and fast memory allocation. Moreover, their Amazon performance test show that browser's probably has much to gain if some sort of heuristic can be used to reuse JITed code between page loads.

The scientists had roughly the similar starting point as I did. They compared benchmarks, tried real web pages, compared with some C++, measured memory consumption and performance. What distinguishes this report from their is that this one focus on the implementations of some applications instead of comparing with real web pages, as comparison to the benchmark tests.

4.3 Comments on Research made by the University of Michigan

Google Chrome browser: a Security and Performance Study is a summary of the researchers findings of how well Google Chrome 1.0.154.35[3] performs compared to other major web browsers [17]. They looked at both the security and performance aspects. The performance was measured by looking at memory consumption, CPU usage, responsiveness, start-up time and JavaScript benchmarks. I'll summarize their findings and specific details of how the measurements were made can be found in the report itself.

[3]When this thesis was written, the latest stable version was 4.0.249.89 and latest beta version was 5.0.335.0.

4.3.1 Memory consumption comparison

For the memory consumption, they first looked at how much memory the browser needs to start up and display several blank tabs. Chrome 1.0.154.35, Firefox 3.0.4, Safari 3.1.2 and Internet Explorer 8 beta was measured. A system with an Intel Core 2 Duo 2.2 GHz with 2 GB RAM running Windows Vista.

Browser	Memory consumption
Chrome (1)	30.5 MB
Chrome (2)	42.4 MB
Firefox	15.9 MB
IE	32.5 MB
Safari	22.4 MB

Table 4.1: Nine blank tabs

(1) means one process per site.
(2) means one process per instance.

Browser	Memory consumption
Chrome (1)	55.9 MB
Chrome (2)	80.6 MB
Firefox	53 MB
IE	33.6 MB
Safari	61 MB

Table 4.2: Google.com opened in two tabs and a new window

Browser	Memory consumption
Chrome (1)	92.3 MB
Chrome (2)	118 MB
Firefox	102.5 MB
IE	118.7 MB
Safari	107 MB

Table 4.3: CNN.com opened in two tabs and a new window

(1) means one process per site. All tabs and windows use the same process.
(2) means one process per site instance. All tabs and windows have their own process.

4.3.2 CPU Usage

They didn't present any data of which the ten web sites they claimed to open were, but instead they stated that Firefox, IE and Safari seemed to consume around 80% of free CPU cycles while Chrome used 100%. The meaning of the statement made is in my opinion useless without any figures and reproducible tests to back them up. The system was an Intel Centrino M laptop with 512 MB RAM running Windows XP Professional.

4.3.3 Responsiveness

The responsiveness of the browsers were tested using a client and server, connected with a wireless network. In short, the client requests a a file, load.html, from the server and a PHP handler records the access time. This is repeated 50 times. The web server was running on a Intel Centrino 1.7 GHz with 512 MB RAM, the client was a Intel Core 2 Duo 2.2 GHz with 2 GB RAM. The server was probably running Windows XP Professional and the client Windows Vista, but no exact information was given.

Browser	Response time	Average
Chrome	20–70 ms	40 ms
Firefox	80–140 ms	90 ms
IE	60–220 ms	105 ms
Safari	40–70 ms	50 ms

Table 4.4: Browser responsiveness. The numbers are taken from the report's graphs 5a and 5b.

These tests have little or no meaning when it comes to testing JavaScript performance because they focus on testing the browsers HTML parsing, network and rendering speed.

4.3.4 Start-up time

The start-up time was measured using a packet sniffer called Ethereal, which listened for the first GET request the browser made. For some peculiar reason, Safari wasn't measured. The system was an Intel Centrino M laptop with 512 MB RAM running Windows XP Professional.

Browser	Right after reboot	Exploiting locality of reference
Chrome	1.5 s	0.4 s
Firefox	0.9 s	1.0 s
IE	1.0 s	0.4 s

Table 4.5: Browser start-up time. The numbers are taken from the report's figure 6.

4.3.5 JavaScript Benchmark

Using an Intel Core 2 Duo 2.2 GHz with 2 GB RAM probably running Windows Vista, the team ran the SunSpider benchmark suite to collect information about how the browsers performed. They found that Chrome vastly outperformed the other browsers, IE by a factor of 13, Firefox by a factor of 7 and Safari by a factor of 2.

	3D	Access	BitOps	Control	Crypto	Date
Chrome	227.2 ms	172.2 ms	134.6 ms	7.6 ms	112.2 ms	333.6 ms
Safari	543.6 ms	670.8 ms	586.4 ms	123.2 ms	385.4 ms	411.6 ms
Firefox	2669.8 ms	1579.6 ms	3574.6 ms	136.6 ms	850.8 ms	2122 ms
IE	1296.2 ms	1937 ms	1791.2 ms	493.6 ms	1098.2 ms	913.2 ms

Table 4.6: SunSpider Benchmarks Figures. The numbers are taken from the report's table 5.

	Math	Regexp	String	Total
Chrome	164.4 ms	330.2 ms	529.2 ms	2011.2 ms
Safari	559.4 ms	358.6 ms	1046.6 ms	4685.6 ms
Firefox	1287 ms	723.6 ms	2349.4 ms	15293.6 ms
IE	1197.6 ms	428.8 ms	17370.4 ms	26526.6 ms

Table 4.7: SunSpider Benchmarks Figures. The numbers are taken from the report's table 5.

4.4 Comments on a Paper from the University of Calgary

Benchmarking Modern Web Browsers confirms the findings of the University of Michigan report [18]. They measure start-up time in the same way the Michigan report does, but with the important distinction that they measure using a local web server rather than an networked one. Additionally they run the SunSpider benchmark to test JavaScript, run render speed tests and finally benchmark the networking performance using AJAX requests. All benchmarks were run on an Intel Core 2 Duo 2.4 GHz (model 6600) with 3 GB of RAM running Windows XP Professional SP2.

The results interesting for this thesis was the start-up time and JavaScript benchmarks.

Browser	JavaScript Engine	Version
Firefox	SpiderMonkey	2.0.0.13 and 3.0.1
IE	JScript	7.0.5730 and 8 (beta)
Opera	linear_b and Futhark	9.26 and 9.51
Safari	JavaScriptCore	3.0.4

Table 4.8: The browsers used in the report

4.4.1 Start-up time

The cold and warm tests were run three times each. A cold start means a recently booted up computer. A warm start means that the browser already have been started several times.

Browser	Cold start	Warm start
Firefox	380 ms	380 ms
IE	350 ms	340 ms
Opera	540 ms	500 ms
Safari	300 ms	300 ms

Table 4.9: Browser start-up time

4.4.2 SunSpider Benchmarks

The numbers have been rounded.

Browser	3D	Access	BitOps	Control	Crypto	Date
Firefox	1.8 ms	1.2 ms	3 ms	0.09 ms	0.6 ms	2 ms
IE	1.2 ms	1.9 ms	1.7 ms	0.5 ms	1 ms	0.8 ms
Opera	0.7 ms	1.15 ms	1.25 ms	0.11 ms	0.6 ms	0.9 ms
Safari	1.1 ms	1.7 ms	1.3 ms	0.11 ms	0.7 ms	0.6 ms

Table 4.10: SunSpider Benchmark Figures

Browser	Math	RegExp	String
Firefox	1.1 ms	0.7 ms	2 ms
IE	1.2 ms	0.25 ms	11 ms
Opera	0.7 ms	0.8 ms	3 ms
Safari	1 ms	0.2 ms	1.2 ms

Table 4.11: SunSpider Benchmark Figures

With Firefox 3.0.1, IE 8 (beta) and Opera 9.51.

Browser	3D	Access	BitOps	Control	Crypto	Date
Firefox	0.3 ms	0.4 ms	0.3 ms	0.04 ms	0.2 ms	0.25 ms
IE	0.9 ms	1.5 ms	1 ms	0.2 ms	0.6 ms	0.7 ms
Opera	0.5 ms	0.6 ms	0.5 ms	0.05 ms	0.3 ms	0.8 ms

Table 4.12: SunSpider Benchmark Figures

Browser	Math	RegExp	String
Firefox	0.35 ms	0.2 ms	0.7 ms
IE	0.8 ms	0.4 ms	2.5 ms
Opera	0.4 ms	0.5 ms	2.5 ms

Table 4.13: SunSpider Benchmark Figures

Chapter 5

Methodology

The specific test and development environment used and what web browsers that were tested, can be found in appendix G (130).

5.1 Method

By way of introduction, I examined literature and academic papers written about JavaScript. These articles could be no more than 10 years old. Older articles were expected to contain errors, mainly because of the tremendous development JavaScript had gone through the past decade. I looked for facts answering my questions, supporting the hypotheses made in the problem formulation. The answers were expected to be different from paper to paper and sometimes even contradict each other.

The next step was to identify suitable scenarios and tasks, implementing them in C++ and JS, identifying differences, similarities and to collect test data. First, larger tasks were tried, like implementing a GIF decoder and an XML parser, primarily to measure performance, memory usage, lines of code and code readability. These tasks were later broken down to smaller tasks and examined individually. The purpose was to identify where, how and why performance was different between JS engines.

After collecting test data from the various implementations, they were compared to results collected from industry benchmarks and applications written by third parties, like the V8 Benchmark Suite and a JavaScript JPEG encoder [19, 20, 5]. Finally, the browsers' JavaScript parsing performance were compared using a Parse 'n Load test suite [21]. It's an important step measuring the overhead associated with a script language, pointing out that not only raw performance matters but also how fast the browsers can respond from initially loading the web page.

All test cases were run against the most recent stable versions of the popular web browsers Chrome, Firefox, Internet Explorer and Opera, available at the end of March, in 2010.

5.1.1 The GIF Image Decoder

First, the GIF decoder was implemented and then tested so it decoded images correctly. Support for GIF image animation was not implemented. Because JavaScript runs within a virtual machine, it can't access the user's file system, and the test images had to be stored as Base64 encoded strings in the source code. That added some parsing, execution and memory overhead, compared to C++ programs reading their input directly from local files. On the other hand, by using this method JavaScript got no I/O overhead from the file system. The first version's performance and memory usage were measured. This initial version used arrays of variable size to store the image data.

After the first round of test data had been collected, the GIF decoder was run through a profiler, specifically, the profiler available in the Firefox extension called Firebug. When bottlenecks had been identified, and minimized or removed, a new round of test data was collected. This step was repeated until a total of four versions had been created and measured. All versions were optimized with the Google Closure Compiler to see if the memory usage could be lowered. The forth version used arrays of fixed size to store the image data.

As a last step of optimization, some JS code was lifted out, implemented in C++ and put in the JS engine. By doing so, I could test my hypothesis that the right chosen JS code rewritten in C++ might change the performance dramatically. The JS engine modified in this thesis was Opera Software's new Carakan engine.

5.1.2 The XML Parser

An object oriented C++ version was first written, tested and optimized. Then the JS version was created borrowing some of the ideas of object abstraction used in the C++ version.

5.1.3 Individual Tasks

Only a few of the tests were chosen for comparison. The tests focus on array access, mathematical calculations, matrix operations, method calls, loops and string concatenation; operations that are important if you want to create desktop-like applications, games or image decoders, to name a few examples. The tests in the Shootout can not be directly compared to the ones performed in SunSpider or V8, but they give a hint of how close the speed of JavaScript is to C++.[1]

[1]See appendix H (page 132) for a detailed description of the tests.

5.1.4 JPEG Encoder

In 2009, developer Andreas Ritter implemented two JPEG encoders in JavaScript. One single-threaded and one using WebWorkers to spawn multiple threads. JavaScript lacks a native byte array, so the developer was forced to try different approaches of how to represent the image's data in a fast and efficient way. Comparing his encoder to the GIF decoder is interesting, mainly because of his different approach to representing the image data, and to see if the JPEG encoder's relative performance distribution matches the GIF decoder's in the browsers tested.

5.1.5 Industry Benchmarks

SunSpider and V8 are the benchmark suites that all major developers of JavaScript engines use to compare performance. Most scientific papers use only them to compare the progress of JavaScript, ignoring use cases the suites misses [20, 19]. [2]

5.1.6 Implementation Method-flow

These steps were used when implementing the GIF decoder and the XML Parser.

1. **Identify and Design**
   Identify what needs to be done and create a suitable design. For example, if an image decoder is to be created, the first step would be to research the image format from its specification and also look into other sources that might contain useful information, to get a deeper understanding of the format.

2. **Exploration of Behavior**
   Files created in the format that is researched will be examined. It's easy to identify misunderstandings or lack of information in the former task by looking closely at how the format behaves. A text or hex editor will be used to assist in the knowledge gathering.

3. **Dividing, Designing and Implementing**
   Larger tasks are divided into smaller ones. By creating an overview of how the tasks fit together, using a UML tool or by drawing by hand, it's easier to stay focused on a single implementation, knowing what that task is expected to deliver and what input it must handle. Next, the tasks will be implemented in an easily predictable high level language, like Python. However, if it's obvious that the tasks as easily can be implemented in JavaScript and C++, it should be done directly.

[2]See appendix H (page 132) for a detailed description of what the test suites do.

The purpose is to quickly implement something, to test if the gathered knowledge and understanding of the tasks were correct, without worrying about memory allocation, pointers or how to solve the problem most effectively.

4. **Test Cases**
   Write test cases for every individual module and their expected combined behavior, that is, the expected behavior the modules are supposed to have while being used together.

5. **Run Tests**
   Run the test cases and fix errors discovered in the implementation.

6. **Implement in C++ and JavaScript**
   Examine if the chosen design holds for implementation in C++ and JavaScript. Modification of the design might be needed, by creating additional objects, classes and helper functions.

7. **Test the Implementation**
   Test the implementation against the test cases. Correct discovered mistakes.

8. **Measure Performance**
   Collect test data by measuring performance using timers, memory consumption, responsiveness etc.

9. **Profiling**
   Profile and optimize the code. Run all the test cases again, correct errors and then collect new test data.

10. **Optional: Rewrite JS parts to C++**
    The JS engine might spend a lot of its time in a limited area of the JS code. Try rewriting these areas to C++ to see if the performance improves significantly.

Step ten is considered optional because it was only used in the GIF decoder implementation.

Chapter 6

Implementation

The implementation of the GIF decoder was far more complex than implementing the XML parser or the elementary tests based on *The Great Win32 Computer Language Shootout*. This is why this section primarily focuses on the decoder rather than the XML parser.

6.1 Implementing an Image Decoder

6.1.1 Purpose

A GIF decoder was implemented because image decoding is the kind of task one normally doesn't associate with a high level language. Most image decoders are implemented in C, C++, Java or some other language that can't be run without being compiled first. The machine code produced by the compiler for a traditional language can be heavily optimized, as the time this takes is of little concern to the end user. A script language on the other hand, like JS or Python, have a limited time frame for optimizations, as the user expects the web page or application to start almost immediately. This implementation was made to measure how close JavaScript can get to the performance of a traditional language, especially when dealing with binary data and mathematical intensive operations.

The specific purpose of implementing an image decoder is to compare the performance of the types *String* and *Array* in JS, compared to C++, that natively can handle binary data. The comparison is especially interesting because JavaScript is primarily designed for the web and has excellent functions to handle String data, using regular expressions and other helper functions, but it lacks byte arrays. An implementation using binary data could potentially give interesting results depending on how the makers of JS engines have chosen to implement strings and arrays. Also, number of code lines, development utilities, memory usage, reusability and how often a typical image decoder is used have been measured.

6.1.2 Choosing a Suitable Format

I wanted a bitmap image format that uses a loss-less data compression, support transparency and color, and is relatively well-known and supported in every web browser. By having native support in a browser for the considered formats, it's easy to compose a simple web page to benchmark the decoding speed of its C++ implementation compared to one written in JS. GIF and PNG were the only candidates that really could be considered as there are no other loss-less formats that fit the requirements.

Compared to GIF, PNG is more complex and advanced. It supports 24-bit colors including 8 bits for transparency. The compression is divided in two phases, first by filtering and finally using DEFLATE-compression. The PNG specification seems better documented, and might be easier to understand as it has graphics explaining many aspects of the format, while the GIF specification is pure text.

After some thought, GIF was considered the best choice because the format isn't too complex to understand, even though the specification could have been better written. The format uses a maximum of 256 colors, supports transparency, has optional animation and uses the LZW compression algorithm to pack the data. LZW is a well known algorithm that's taught to students in courses introducing data compression, like in the course TSBK08 at Linköping University. To make the implementation easier, I read through the specification and picked out only the most important parts, clarified them and put the information in a new document. [1]

6.1.3 Architecture and Design

The first attempt to implement a GIF decoder failed, primarily because there was no architectural thought behind it [2]. The second attempt used an object oriented model in which I saw the GIF file as a large object with sub-objects. Two major sub-objects were created with one handling LZW data decompression and another for storing all the GIF specific data, like color tables, image size, transparency etc. Objects were then thought out for the data structures needed. Each object was designed with loose coupling in mind, so everything could be tested individually before connecting them together. Self tests were created for every object with the purpose of making it trivial to catch introduced bugs after changing the code.

[1]This document is found in appendix C (page 87).

[2]Details about this can be found in appendix I (page 136).

6.1.4 Implementation

Before implementing the object oriented design, I first took some LZW data and wrote a small decompressor in Python. By using Python, it was possible to focus entirely on the task at hand, as all the data structures and other useful help functions were already there, as part of the programming language. [3]

When I felt confident that I've understood the concept of LZW decoding, I began writing an object oriented GIF parser in C++. First, I implemented the objects, data structures and help functions needed to work with them. Second, various tests were written to be able to point out exactly where errors occurred. If some function later needs to be changed, the tests can be run to quickly verify the consistency of the function's behavior. Third, the parsing of the grammar described in appendix C were implemented. I opened up a GIF file[4] in an HEX editor and compared the file's expected behavior with the output from my decoder. Last, LZW was implemented.

This implementation turned out to perform equally or worse, compared to the GIF decoders available in the examined web browsers. Because of that, I chose to abandon my decoder in favor of the built-in ones, as their performance was similar and had more mature code. To measure the browsers decoding speed, I created a small web page that used some JavaScript to time the loading of a chosen image.[5]

Next, I implemented a GIF decoder in JavaScript. It's heavily based on Opera's own C++ GIF decoder, but rewritten to suit JavaScript. Code lines that was unnecessary for the problem formulation were removed. For example, the code handling GIF animation was removed. Rewriting large amounts of C++ to JavaScript is time consuming, so several patterns were written to speed up this conversion. Error handling was improved, variable declarations changed and memory allocation removed, to name some changes.[6]

Initially, I thought that the conversion would take a lot of time, even with the patterns. Luckily, I was wrong. After about two weeks of coding and testing the decoder was partly working, being able to decode black and white images, like figure 12.1. The decoder was far from perfect, and still had trouble decoding larger images, especially with color or transparency. The hardest part was not to decode the GIF stream, but to make sure the the LZW decoding did exactly what it was expected to do, and to pass the image information along to a structure that could show it on a web page with the correct colors. Before every step of the decoding and rendering were correct, there were several decoding mistakes like the one shown in

[3]The source code of the decompressor is found in appendix A (page 82).

[4]See figure 12.1 (page 83).

[5]The code is found in appendix E (page 115).

[6]The conversion patterns are available in appendix F (page 117).

figure 12.6. In the end, the decoder were able to decode images even as big as figure 12.5.[7]

6.1.5 How Often is the GIF Decoder Used?

A normal web page consist of at least one picture, often more, 10–50 isn't unusual. A regular user expects that the page loads fast, and that the web browser can handle multiple tabs without being slowed down. Knowing this, the image decoder had to be both memory efficient and fast, especially if used on mobile devices, because the code can be requested hundreds of times every minute.

6.1.6 Development Utilities

At the start, then I was implementing the GIF C decoder, I was using a Windows 7 environment together with Visual Studio. Visual Studio was really helpful in helping debug pointers and memory leaks, all integrated in the same environment I wrote my code in, letting me easily pinpoint the exact line of the issue. Even though the architectural thought was non-existent, I felt very productive, because the errors could quickly be solved and everything *just worked*. In my opinion, tools visualizing and gathering the compiler, editor and debugger in the same IDE makes a big difference in productivity compared to cumbersome command line utilities.

Before implementation the C++ version, I moved to an Ubuntu Linux[8] system. First, I tried using Eclipse as IDE but I really didn't like it that much. The interface was sometimes fast, sometimes slow, making it very irritating and unpredictable to work with. Compilation worked within the IDE, while debugging didn't. If I'd spent time trying to solve the debugging issue Eclipse might have been more useful, but instead I switched to KDevelop. KDevelop had an automated build system using QMake, which I however, chose to not use. Instead, I wrote my own build script and used Valgrind together with KCachegrind to visualize calls and do profiling. What really impressed me with KDevelop was the powerful built-in auto completion of C++ code. Visual Studio have the same feature, but KDevelop's performance and almost non-exist lag made it far superior in this task. To debug memory leaks and errors in a Linux environment using GDB can be cumbersome, but KDevelop have an integrated debugger making the debugging almost as easy as in Visual Studio.

The development utilities available for JavaScript lacks many of the features a C++ programmer would expect. To my knowledge, there doesn't

[7]Most JS parsers experienced problems while decoding figure 12.5 (page 85) because of the long string concatenation it required.

[8]Ubuntu Linux 9.10

exist a single IDE combining both code completion, syntax highlighting, debugging, lint checking and profiling. Eclipse offers most of this, but lacks a debugger, profiler and the code completion sometimes forgets declared variables and functions.

When working in Windows, I preferred using the editor Notepad++ together with the Firefox Firebug extension as profiler and debugger, and jslint.com as lint checker. Chrome and Opera also supplies utilities for web developers, for network call benchmarking, debugging and profiling. Opera's Dragonfly is especially good at letting the developer step the JavaScript code, line by line. In Linux, I used the same tools but with KDevelop as editor.

6.1.7 Discovered Problems Along the Way

Having an architecture and a well-planned design is vital to successfully implement a solution to a problem. Without it, a lot of time risk being spent on rewriting and reevaluating the approach, or worse, discovering that the you solved a another problem than the intended. A systematic approach, thoroughly done research and using lots of test cases saved me weeks of debugging time, simply because I didn't get that many bugs and my focus was aimed at solving the right problem.

Opera doesn't use the Standard Template Library in C++, which forced me to implement my own tables, dictionaries and other needed data structures. This was time consuming but educational, as I had to repeat my knowledge of data structures, algorithms, memory allocation and pointers.

Debugging pointers were sometimes extremely time consuming, until I started using assert functions at the start of every function and before important lines of code within a function. Creating effective assert statements are very much like writing good test cases. They take a while to write, but it's well spent time. When I began using assert statements my time spent on debugging almost disappeared. The code became extremely predictable.

Like pointer errors, there were also issues with memory allocation. Allocation too little memory, forgetting to allocate new memory, freeing memory etc. This was solved using assert statements and the memory utility Valgrind.

Before realizing the importance of test cases and assert statements, I spent weeks of time focusing on small errors that really shouldn't have been an issue. Instead of focusing on the real problem, I had to solve surrounding problems, like issues with my implementations of color tables and dictionaries.

6.1.8 Profiling and Optimization

When the first implementation of the JavaScript GIF decoder had been written and tested, the code was profiled to identify potential performance issues.

Look at the first seven lines in the table below (page 44). The functions' are responsible for decoding LZW data, to link the right code word with its corresponding color in the color table, to add new unknown code words to the decoding table and finally deliver the decoded data to a image object. These functions make up for 76% of the applications run time. Many are called repeatedly, more than 50 000 times per execution. Small optimizations can therefore make a big impact on performance.

The first attempt at this was to look into the possibility of minimizing array access, as look-ups are expensive. In version 1, OutputString adds code words directly to an array without first looking up its corresponding color. The color look-up is performed later, in OnOutputAll, forcing it to iterate through all the code words again. Instead, it's possible to access a chosen value in an array and store it as a local variable. In theory, accessing a local variable is much faster than having to perform a repeated array look-up. This idea was tested, by creating version 2 of the decoder. The color look up was moved from Inaptly to OutputString, increasing its operation time from 414 to 1194 ms. OnOutputAll, on the other hand, now performs below 1 ms, compared to 699 ms. As a final change, two helper functions were added to LzwStringTable, one to look up the right color table and another to get the object where the decoded data is stored. In total, the optimizations save 8 lines of code and lower the operation time in the affected functions by about 100 ms. However, the total performance decreased.

In version 3, the changes made in version 2, were mostly reverted. The look-up of colors was moved to the function creating the color table. It makes no sense to first create a color table, look up a code's color and finally converting it to another representation. With this change, a step can be skipped. Functions with lots of array access were changed to use more local variables. As an example, AddString's array look-ups were cut down from seven to two, increasing the functions performance by 50%. Most assert functions were commented out, as they were only needed for debugging. In total, these optimizations increased the decoding performance by 53–135% and 63–150% compared to version 1 and 2.[9]

As a last modification, version 4 got its color representation changed. Before, the colors were stored in separate channels, red, green, blue and alpha. They couldn't be added together and displayed without additional mathematical operations, because without them, the image became distorted by the wrong positioning of the bits, which figure 12.6 and 12.4 show. The colors are still stored in separate channels, but shifted to allow easy addi-

[9]See table 6.1 (page 43).

tion. This modification decreased the performance slightly in Epiphany and Opera, while marginally increasing it in Chrome and Firefox, perhaps because of the somewhat fewer lines of code. N.B.: The change of performance falls within the margin of error.

When comparing the four versions and their profiling output, it's clear that the relative time spent in DecodeCode() increases, while the number of calls remains the same. To optimize further, the most expensive functions can be moved inside the JavaScript engine to additionally boost the performance. As it turned out, that meant moving all LZW decoding inside the engine, increasing the total performance by an additionally 100–200%.

Only functions that are used more than 0.05% of the time is shown in the tables below. Firefox Firebug extension was used to collect all profiling data.

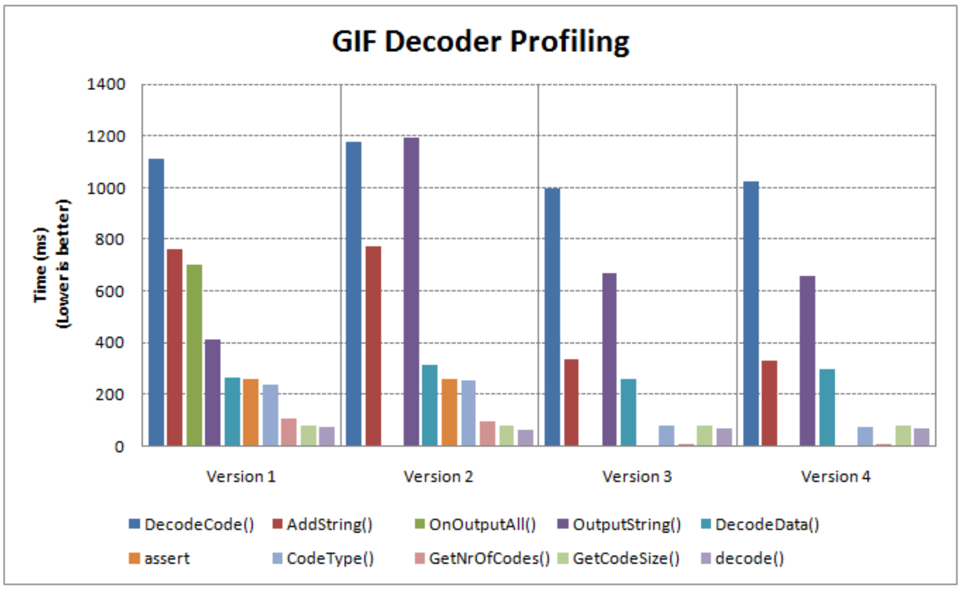


Figure 6.1: Firebug profiling of all GIF decoder versions

Version 1

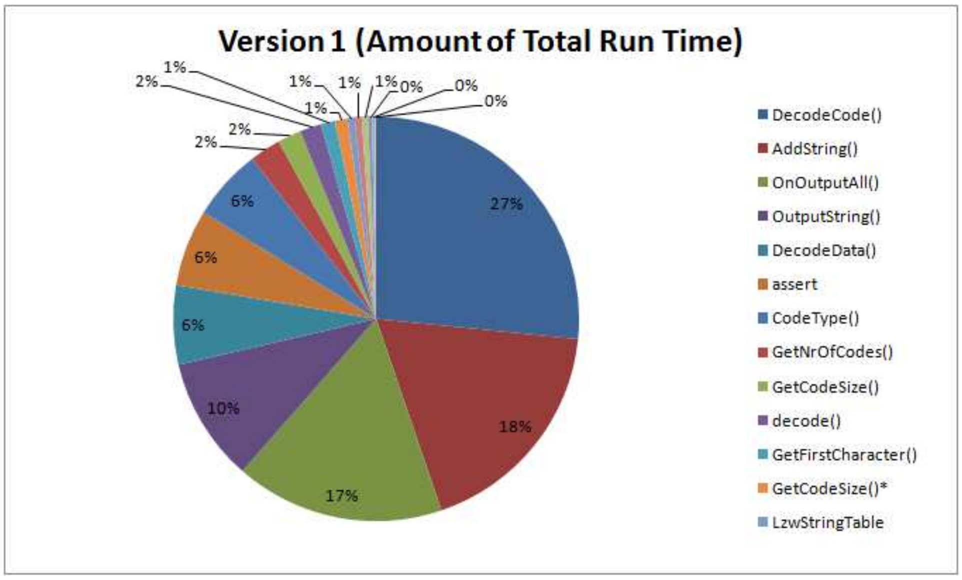


Figure 6.2: Firebug profiling of GIF decoder version 1

Function	Calls	Percent	Own time	Time	Average	Line
DecodeCode()	56200	22.57%	1113 ms	3652 ms	0.065 ms	2261
AddString()	56169	15.43%	761 ms	1026 ms	0.018 ms	2026
OnOutputAll()	1	14.17%	699 ms	699 ms	699 ms	2539
OutputString()	56184	8.40%	414 ms	414 ms	0.007 ms	2089
DecodeData()	312	5.36%	264 ms	4038 ms	13 ms	2416
assert	395047	5.21%	257 ms	257 ms	0.001 ms	3321
CodeType()	112369	4.85%	239 ms	313 ms	0.003 ms	2056
GetNrOfCodes()	57649	2.12%	104 ms	141 ms	0.002 ms	2145
GetCodeSize()	56512	1.60%	79 ms	121 ms	0.002 ms	2355
decode()	1	1.46%	72 ms	72 ms	72 ms	1524
GetFirstCharacter()	56169	0.95%	47 ms	47 ms	0.001 ms	2129
GetCodeSize()	56512	0.85%	42 ms	42 ms	0.001 ms	2137
LzwStringTable	1	0.48%	24 ms	46 ms	46 ms	1971
code()	4096	0.46%	23 ms	23 ms	0.006 ms	1705
Clear()	16	0.44%	21 ms	21 ms	1 ms	2000
alert_w	313	0.27%	13 ms	13 ms	0.043 ms	1503
LzwDecoder	1	0.20%	10 ms	56 ms	56 ms	2399
DecodeDataInternal()	1	0.07%	4 ms	4113 ms	4113 ms	2692

Table 6.1: Profiling output of version 1 (imagedecodergif.js)

Version 2

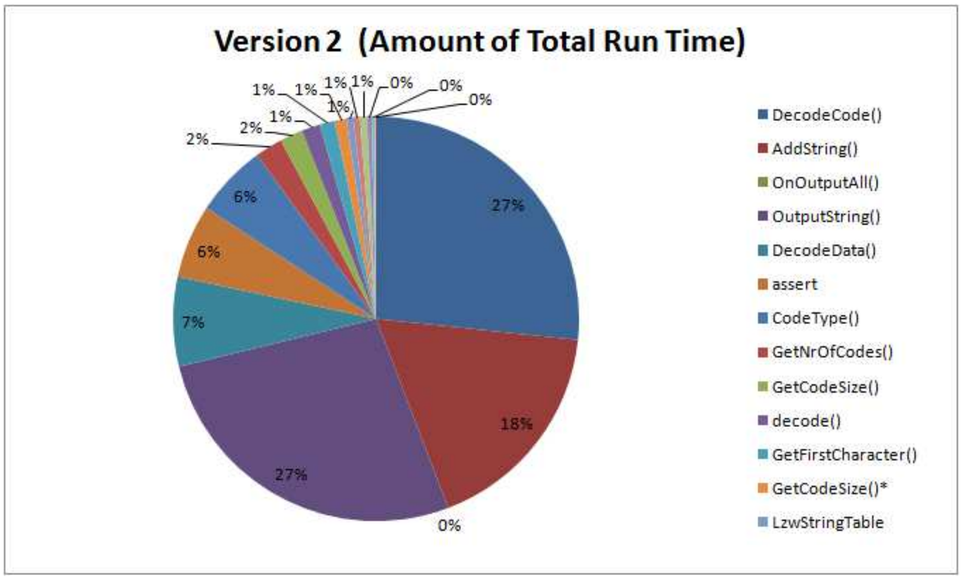


Figure 6.3: Firebug profiling of GIF decoder version 2

Function	Calls	Percent	Own time	Time	Average	Line
OutputString()	56184	22.76%	1194 ms	1195 ms	0.021 ms	2089
DecodeCode()	56200	22.46%	1178 ms	3837 ms	0.068 ms	2276
AddString()	56169	14.78%	775 ms	1046 ms	0.019 ms	2026
DecodeData()	312	6.00%	315 ms	4275 ms	14 ms	2431
assert	395047	4.94%	259 ms	259 ms	0.001 ms	3311
CodeType()	112369	4.88%	256 ms	331 ms	0.003 ms	2056
GetNrOfCodes()	57649	1.85%	97 ms	134 ms	0.002 ms	2158
GetCodeSize()	56512	1.55%	81 ms	123 ms	0.002 ms	2370
decode()	1	1.24%	65 ms	65 ms	65 ms	1524
GetFirstCharacter()	56169	1.03%	54 ms	54 ms	0.001 ms	2142
GetCodeSize()	56512	0.79%	41 ms	41 ms	0.001 ms	2150
Clear()	16	0.48%	25 ms	25 ms	2 ms	2000
LzwStringTable	1	0.45%	24 ms	46 ms	46 ms	1971
code()	4096	0.43%	23 ms	23 ms	0.006 ms	1705
alert_w	313	0.33%	17 ms	17 ms	0.054 ms	1503
LzwDecoder	1	0.18%	10 ms	56 ms	56 ms	2414
DecodeDataInternal()	1	0.09%	5 ms	4355 ms	4355 ms	2669

Table 6.2: Profiling output of version 2 (imagedecodergif.js)

Version 3

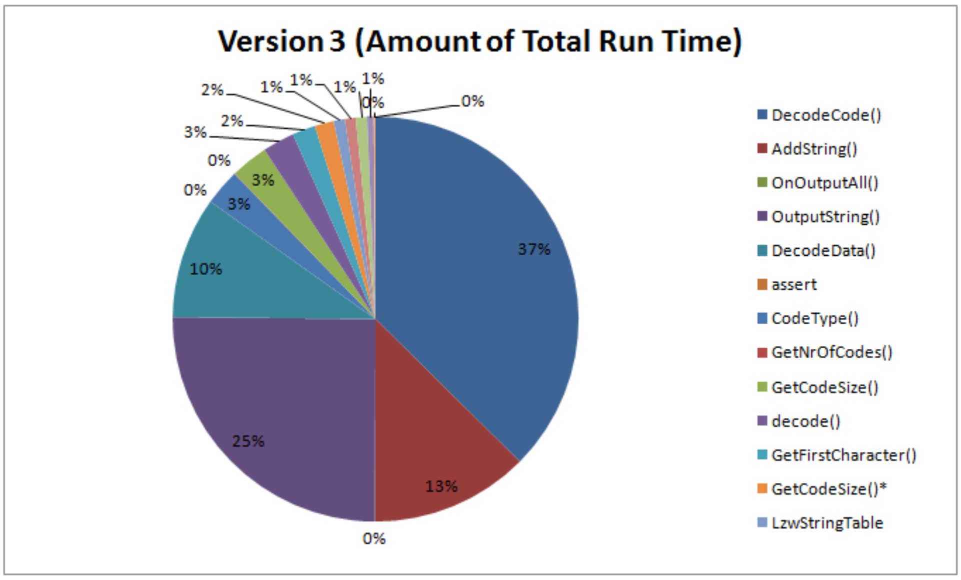


Figure 6.4: Firebug profiling of GIF decoder version 3

Function	Calls	Percent	Own time	Time	Average	Line
DecodeCode()	56200	28.25%	998 ms	2155 ms	0.038 ms	2277
OutputString()	56184	18.98%	670 ms	670 ms	0.012 ms	2095
AddString()	56169	9.56%	338 ms	338 ms	0.006 ms	2026
DecodeData()	312	7.31%	258 ms	2534 ms	8 ms	2432
GetCodeSize()	56512	2.26%	80 ms	121 ms	0.002 ms	2371
CodeType()	56200	2.19%	77 ms	77 ms	0.001 ms	2062
decode()	1	1.93%	68 ms	68 ms	68 ms	1524
GetFirstCharacter()	56169	1.39%	49 ms	49 ms	0.001 ms	2143
GetCodeSize()	56512	1.15%	41 ms	41 ms	0.001 ms	2151
LzwStringTable	1	0.69%	24 ms	47 ms	47 ms	1971
Clear()	16	0.68%	24 ms	24 ms	2 ms	2000
code()	4096	0.64%	23 ms	23 ms	0.006 ms	1705
alert_w	313	0.38%	14 ms	14 ms	0.043 ms	1503
DecodeDataInternal()	1	0.12%	4 ms	2600 ms	2600 ms	2670
GetNrOfCodes()	1449	0.03%	1 ms	1 ms	0.001 ms	2159

Table 6.3: Profiling output of version 3 (imagedecodergif.js)

Version 4

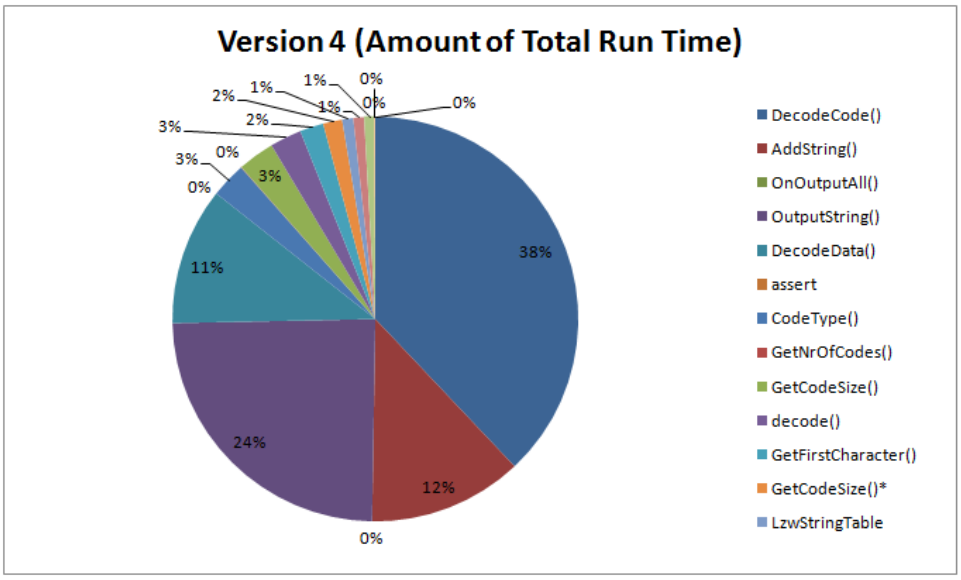


Figure 6.5: Firebug profiling of GIF decoder version 4

Function	Calls	Percent	Own time	Time	Average	Line
DecodeCode()	56200	36.20%	1022 ms	2160 ms	0.038 ms	2267
OutputString()	56184	23.30%	658 ms	658 ms	0.012 ms	2087
AddString()	56169	11.75%	332 ms	332 ms	0.006 ms	2018
DecodeData()	312	10.45%	295 ms	2575 ms	8 ms	2422
GetCodeSize()	56512	2.80%	79 ms	120 ms	0.002 ms	2361
CodeType()	56200	2.71%	76 ms	76 ms	0.001 ms	2054
decode()	1	2.41%	68 ms	68 ms	68 ms	1516
GetFirstCharacter()	56169	1.81%	51 ms	51 ms	0.001 ms	2133
GetCodeSize()	56512	1.45%	41 ms	41 ms	0.001 ms	2141
LzwStringTable	1	0.85%	24 ms	47 ms	47 ms	1963
code()	4096	0.81%	23 ms	23 ms	0.006 ms	1697
Clear()	16	0.78%	22 ms	22 ms	1 ms	1992
DecodeDataInternal()	1	0.06%	2 ms	2625 ms	2625 ms	2660
GetNrOfCodes()	1449	0.04%	1 ms	1 ms	0.001 ms	2149

Table 6.4: Profiling output of version 4 (imagedecodergif.js)

6.2 Parsing XML

6.2.1 Purpose

The implementation of the parser was motivated by comparing how C++ and JS, a language designed to manipulate strings, handles large amounts of non-binary string data. Especially, what performance and amount of code that were needed to parse XML like documents.

6.2.2 Choosing a Suitable Format

The parser implemented isn't a *real* XML parser, but can be seen as a subset of real XML. It was written to parse iso-8859-1 encoded documents made out of XHTML. Specifically, the HTML document describing the *XHTML 1.0: The Extensible HyperText Markup Language (Second Edition)* specification was used to test the parser.

The reason only a subset of XML was chosen to be supported was because that made the JS and C++ implementation much easier. Without having to care about special characters, regular expressions could be used to speed up the parsing.

6.2.3 Architecture and Design

The HTML document describing the specification was analyzed and a list of potential objects were made. Elements, attributes, comments and others were divided into own objects that could easily be implemented one by one, and then tested.

6.2.4 Implementation

After the object model was created, a C++ parser was written. This first version had difficulties parsing larger documents, missed attributes deeply nested in the XML, missed elements formatted in a certain way etc. The parser was rewritten but using the same object model. The new version worked much better and also saw a small speed improvement.

The C++ code acted as model for creating the JavaScript version. The majority of the code and its help-functions were replaced with short regular expressions. Some JavaScript engines compile these expressions into efficient machine code which potentially makes some operations in JavaScript faster than in C++.

6.2.5 How Often is the XML Parser Used?

If the use case is to parse XML within a web browser, like browsing web sites, then the parser is executed when visiting new web pages.

6.2.6 Development Utilities

See the same section for the GIF decoder.

6.2.7 Profiling and Optimization

A part from rewriting the parser no specific optimizations were made to the C++ version. However, the JavaScript parser saw a dramatic increase in speed when switching from String comparison to regular expressions. Additionally, many == were changed to ===, to speed up comparisons. This means that instead of doing a deep comparison, like matching both data value and object type, only the highest abstraction level gets compared, like comparing if integer 1 is equal to 2.

6.3 Low-level Tasks

6.3.1 Purpose

All industry benchmarks run specifically only JavaScript code to compare performance, never comparing with running machine code, like compiled C++ code. Because of this, test cases based on JScript code from *The Great Win32 Computer Language Shootout* were ported to JavaScript and compared against their C/C++ counterparts, to give a comparison how well JavaScript engines perform compared to C/C++ compiled machine code [22].

6.3.2 Architecture and Design

The source code for these tasks are heavily based on the source code from *The Great Win32 Computer Language Shootout*, specifically the GCC (C language) and JScript versions.

6.3.3 Implementation

The JScript source code had to be modified to be compatible JavaScript, but most of the original source code is intact. The GCC versions were not touched at all, apart from minor error correction.

6.3.4 Profiling and Optimization

No profiling or optimization of the source code has been made.

Chapter 7

Results

7.1 Motivation

To measure the real average performance of a programming language is very hard. It's not enough to only look at how the language performs in mathematical operations, array and string operations, parsing binary data or regular expressions. Those kinds of tests only simulate a specific and synthetic assumption of the real world, and often only run for a short period of time. That's the kind of tests most scientific papers use, benchmark suites like SunSpider or V8, to evaluate if JavaScript engines have become faster, never comparing their operations against other languages like C++.

In this paper, the focus have been to measure the performance implications of JavaScript on real software, and compare that to equivalent C++ implementations. Not just short JavaScript only theoretical tests but implementations of real world software. To do this, a GIF decoder and an XML parser were written, software that combines lots of functions, binary parsing, looping, mathematical calculations etc. The JavaScript decoder can for example be used instead of the web browser's native decoder. The XML parser is excellent for letting a web application parse XML like data on the client, especially in situations where JSON isn't suitable.

Additionally, a similar program has been included, a JPEG encoder. It was included to have something to compare to the GIF decoder. This software lets web applications move their CPU burden from the server to the users' computers.

In addition to the implementations mentioned, some basic tests were written, similar to those found in SunSpider and V8. They were written in both JavaScript and C++.

7.2 Outline of the Results

The results begin with a section called *Interesting Performance Results*, with a graph displaying the historic performance development of JavaScript engines found in various web browsers. Next, the performance of the JPEG encoder and the GIF decoder are presented followed by a lines of code comparison and last a graph displaying memory usage. The results show what kind of performance, number of code lines and memory usage one could expect from typical software parsing binary data and performing mathematical calculations implemented in JavaScript. The results that then follows displays the performance and lines of code required to implement a kind of XML parser.

The *Time* section presents the time it took to do the various implementations. It's followed by the next section, *Multitasking and Responsiveness*, that shows an often forgotten aspect of JavaScript: multitasking and the time it takes for the engine to load, interpret and start running the scripts. To be an ideal platform for running web based applications, a browser can't lock all of its resources to one script at a time. It must handle concurrent operations in a fair responsive manner.

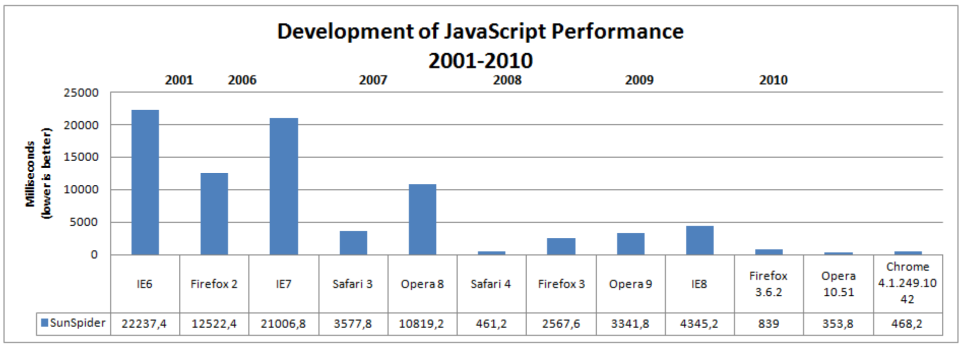
Before presenting the results from running the test suites made by Apple and Google, a section called *Elementary Tests* compare the performance of some basic operations run in both JS and C++. These are similar to many found in the industry benchmarks, but differ in that they compare their performance with another language. Last, in *Test Suites*, results from running the well-known industry benchmarks SunSpider and V8 are presented. They've been included for easier comparison to other scientific papers.

If not stated otherwise, all tests were performed several times on the same computer, a laptop with a 2.5 GHz Intel Core 2 Duo CPU with 4 GB RAM, running Windows 7 Ultimate and Ubuntu Linux 9.10. The power plan was set to *High Performance.* To minimize measurement errors, as few applications as possible were running at the same time as the benchmarks. Only the web site needed to perform the benchmark was open in the browser.

N.B. In browsers, only JavaScript performance and memory usage is presented. Rendering speed and other specific web browser factors have been ignored. A value of 0 means that the browser couldn't run the test.

7.3 Interesting Performance Results

7.3.1 Historic Development

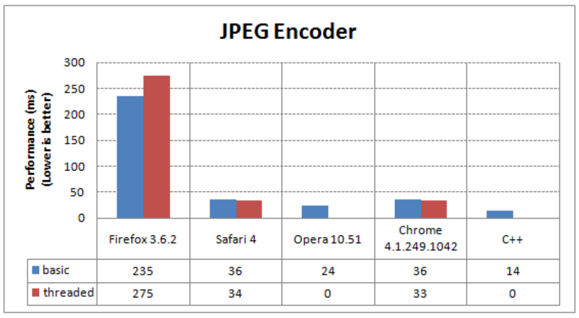


	IE6	Firefox 2	IE7	Safari 3	Opera 8	Safari 4	Firefox 3	Opera 9	IE8	Firefox 3.6.2	Opera 10.51	Chrome 4.1.249.1042
■SunSpider	22237,4	12522,4	21006,8	3577,8	10819,2	461,2	2567,6	3341,8	4345,2	839	353,8	468,2

Figure 7.1: Performance Time Line of JavaScript

Note how the performance increases a lot around 2007–2008.

7.3.2 JPEG encoding



	Firefox 3.6.2	Safari 4	Opera 10.51	Chrome 4.1.249.1042	C++
■ basic	235	36	24	36	14
■ threaded	275	34	0	33	0

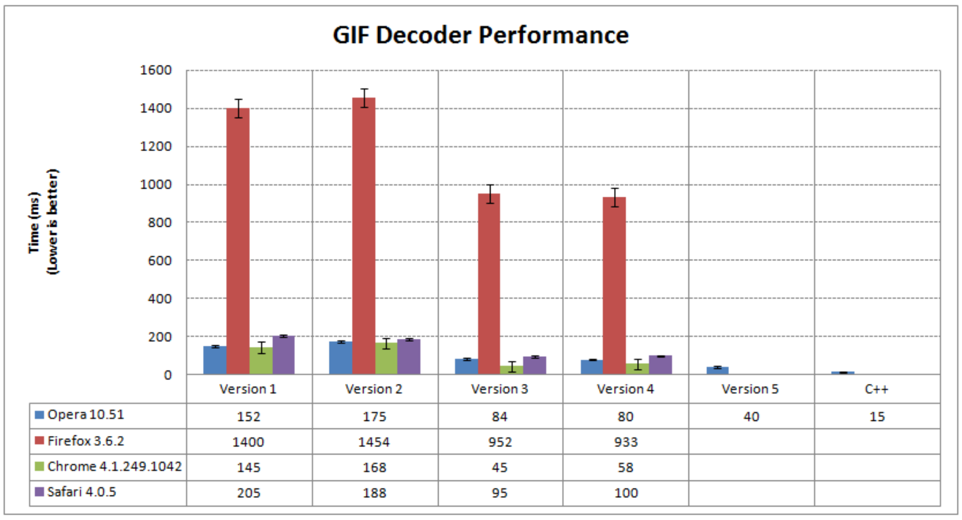
Figure 7.2: JPEG Encoder, non-threaded and threaded

The image used for this encoding benchmark was figure 12.3.1 (page 86). C++ is represented by the time it took for ImageMagick's *convert* utility to encode the test file to JPEG.

Pay special attention to the fact that **Opera runs this test just 1.7 times slower than C++**, and that C++ code can be heavily optimized towards specific platforms using in-line assembler. The JavaScript JIT engine has to be able to optimize *well enough*, within a strict time frame on all platforms and architectures it runs on.

Opera couldn't run the threaded version because at the time of testing, it lacked a working implementation of WebWorkers.

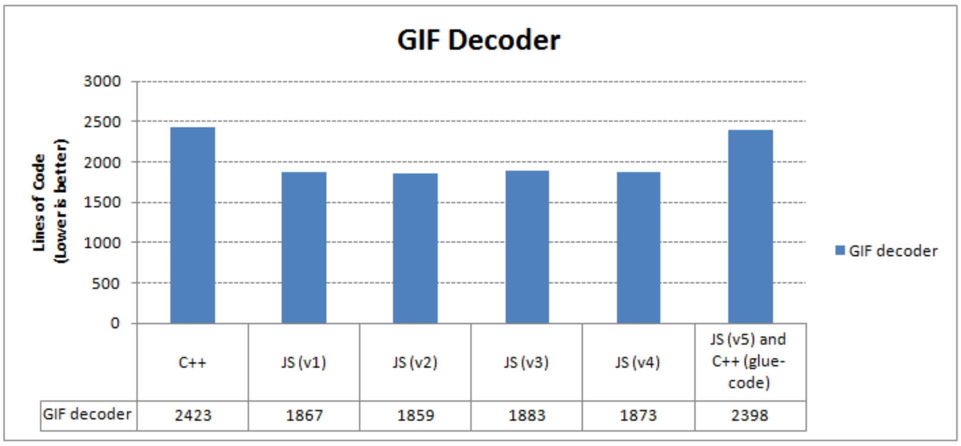
7.3.3 GIF decoding



	Version 1	Version 2	Version 3	Version 4	Version 5	C++
🟦 Opera 10.51	152	175	84	80	40	15
🟥 Firefox 3.6.2	1400	1454	952	933		
🟩 Chrome 4.1.249.1042	145	168	45	58		
🟪 Safari 4.0.5	205	188	95	100		

Figure 7.3: GIF decoding Performance

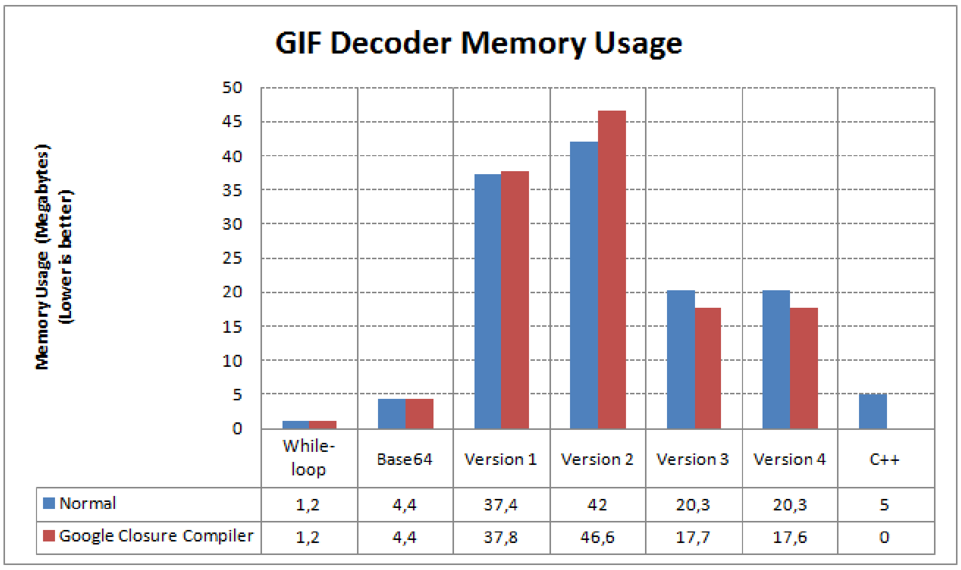
The image used for this decoding benchmark is shown in figure 12.4 (page 84). Version 1-4 are JavaScript only decoders and version 5 is a decoder where the LZW decoding has been put inside the JavaScript engine to offload the script. **Chrome ran version 3 only three times slower than C++, making its JIT engine perform very close to the performance seen in version 5**.



	C++	JS (v1)	JS (v2)	JS (v3)	JS (v4)	JS (v5) and C++ (glue-code)
GIF decoder	2423	1867	1859	1883	1873	2398

Figure 7.4: GIF decoder, Lines of Code

Performance comes at the cost of more lines of code. Is **500 lines of code and only 5 milliseconds faster performance** worth the development time?



	While-loop	Base64	Version 1	Version 2	Version 3	Version 4	C++
🟦 Normal	1,2	4,4	37,4	42	20,3	20,3	5
🟥 Google Closure Compiler	1,2	4,4	37,8	46,6	17,7	17,6	0

Figure 7.5: GIF decoding Memory Usage

Memory usage measured on Opera's Carakan JavaScript engine. The image used for this memory benchmark was figure 12.4 (page 84). The C++ memory usage is an estimation. **Version 3 and 4 consume about 3.6 times more memory than C++.** This is because of how the overhead that's required to make the JavaScript engine handle dynamic string and array types, do type checking etc.

While-loop means the memory consumption of an empty file. *Base64* means the memory consumption of a Base64 decode function and the figures 12.1, 12.2, 12.3, 12.4 (page 83) as Base64-encoded strings. The memory consumption for *Version 1-4* includes the Base64 decoder and the encoded files.

7.3.4 Working with Strings and Running Regular Expressions

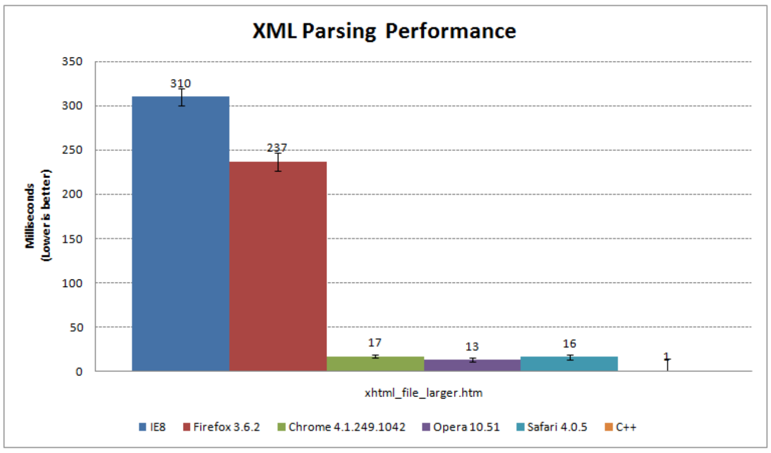


Figure 7.6: XML Parsing Performance

The document used for this benchmark was the *XHTML 1.0: The Extensible HyperText Markup Language (Second Edition)* specification. Note that the error margin is 13 for the C++ version, placing it very close to the performance of the version running in Opera. The large error margin comes from the fact that the performance differed, depending on if it was run in Linux or Windows. The Linux version ran in 1-2 seconds, the Windows-version took 13-14 seconds. This means that **under certain circumstances, Opera runs the JS version as fast as the one written in C++**.

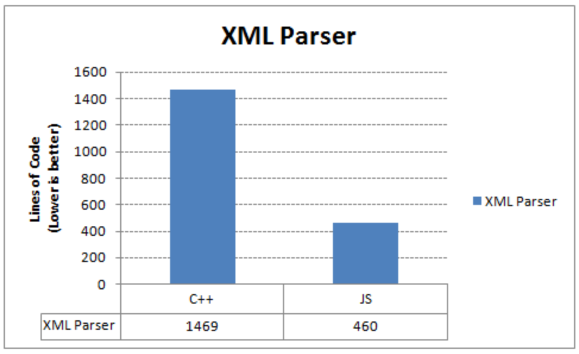


Figure 7.7: XML Parser, Lines of Code

Under certain circumstances the **three times smaller JS code runs as fast as C++**.

7.4 Time

7.4.1 Time Spent on Rewriting the C++ Code to JavaScript

Step	Task	Time
1	Attempted to write a GIF decoder in C without any specific design	5 weeks
2	LZW decompressor written in Python	1 day
3	Object oriented C++ GIF decoder	3 weeks
4	Partly working JS GIF decoder	2 weeks
5	Correcting bugs in the JS implementation	2 weeks
6	Optimizing the JS code	2 weeks
7	Collecting and evaluating the results	1 week

Table 7.1: Time spent on implementing the JavaScript GIF decoder

7.4.2 Time Spent on Writing a XML Parser

Step	Task	Time
1	Created an object oriented model for the parser.	2 days
2	Wrote the first version of the XML parser in C++.	1 week
3	Wrote a more efficient and correct version of the parser.	3 days
4	Corrected bugs within the parser.	2 week
5	Wrote a parser in JavaScript.	3 days

Table 7.2: Time Spent on Writing a XML Parser

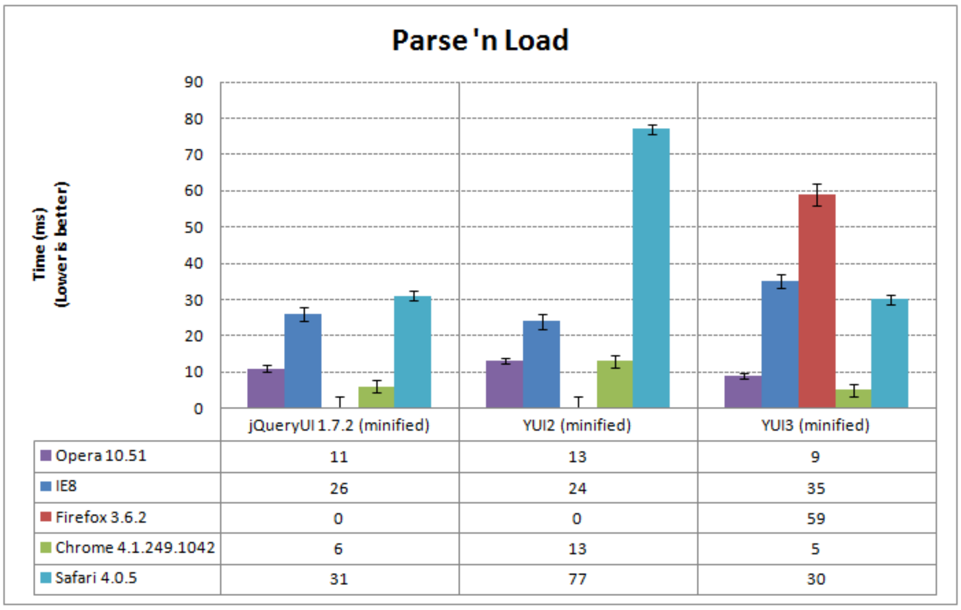
7.4.3 Time Spent on Modifying the Shootout Source Code

Step	Task	Time
1	Porting JScript to JavaScript	2 days
2	Correcting C/C++ compile errors.	1 day
3	Collecting benchmark data.	1 day

Table 7.3: Time Spent on Writing a XML Parser

7.5 Multitasking and Responsiveness

7.5.1 Parse ’n Load



	jQueryUI 1.7.2 (minified)	YUI2 (minified)	YUI3 (minified)
🟪 Opera 10.51	11	13	9
🟦 IE8	26	24	35
🟥 Firefox 3.6.2	0	0	59
🟩 Chrome 4.1.249.1042	6	13	5
🟦 Safari 4.0.5	31	77	30

Figure 7.8: Using Parse ’n Load, 1000 iterations of loading jQueryUI (1), YUI2 (2) and YUI3 (3) were run. jQueryUI, YUI2 and YUI3 are popular JavaScript libraries used by developers all over the world to ease web site development.

For some unknown reason, Firefox couldn’t run the jQueryUI and YUI2 tests.

(1) Version 1.7.2, 379 KB, minified
(2) 390 KB, minified
(3) 311 KB, minified

7.5.2 Cooperative multitasking

While the Parse ’n Load test was running, the responsiveness of the web page was tested by clicking on the list box and by trying to change the text in the iteration text box. The perceived experience was divided into the following categories.

- Unresponsive
- Somewhat responsive
- Responsive

The first means a locked user interface with no responsiveness at all. The second means a slow but still somewhat responsive web page. The third means that the page behaves like it does without any particular load, which is the behavior all web browsers should ideally have.

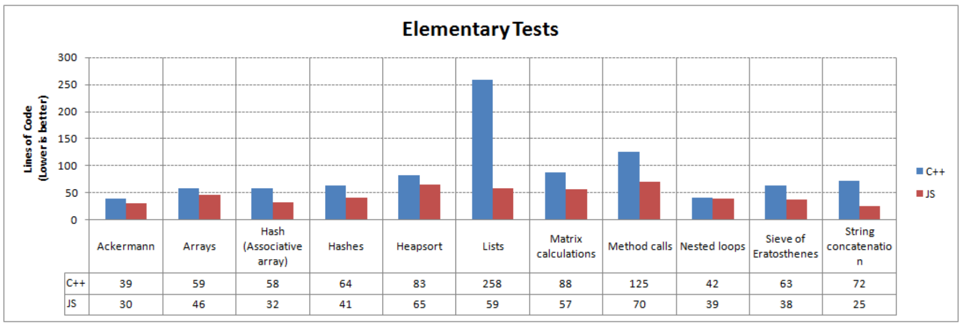
Browser	Responsiveness
Chrome	Responsive
Firefox	Somewhat responsive
IE	Responsive
Opera	Responsive
Safari	Somewhat responsive

Table 7.4: Responsiveness was tested while running the SunSpider Benchmark

What does this mean? Consider that a user has multiple tabs open running different JavaScript based web applications, perhaps a mail client, a Word processing application and some regular web pages. If one such application locks up the whole browser during an intensive operation, that's unacceptable. It can easily be avoided using threads or a fixed amount of CPU cycles allocated to each script.

N.B. Before version 3.6 was released, this test was very unresponsive in Firefox.

7.6 Elementary Tests



	Ackermann	Arrays	Hash (Associative array)	Hashes	Heapsort	Lists	Matrix calculations	Method calls	Nested loops	Sieve of Eratosthenes	String concatenation
C++	39	59	58	64	83	258	88	125	42	63	72
JS	30	46	32	41	65	59	57	70	39	38	25

Figure 7.9: Elementary Tests, Lines of Code

Below, all graphs show the time (milliseconds) it took to perform the tests.

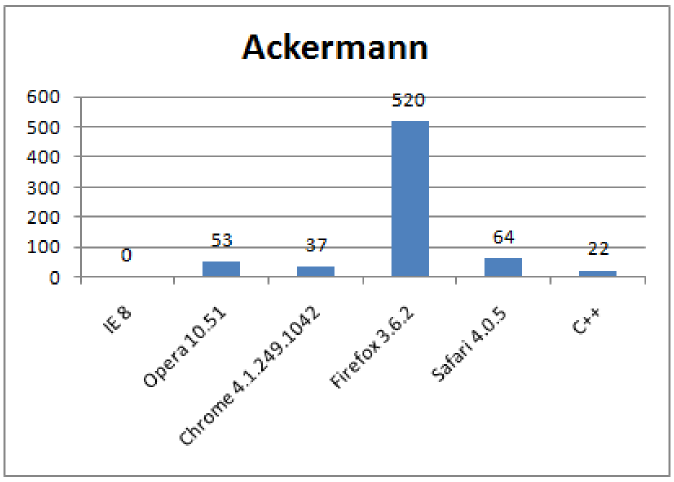


Figure 7.10: The Ackermann function is a simple example of a computable function that's not primitive recursive

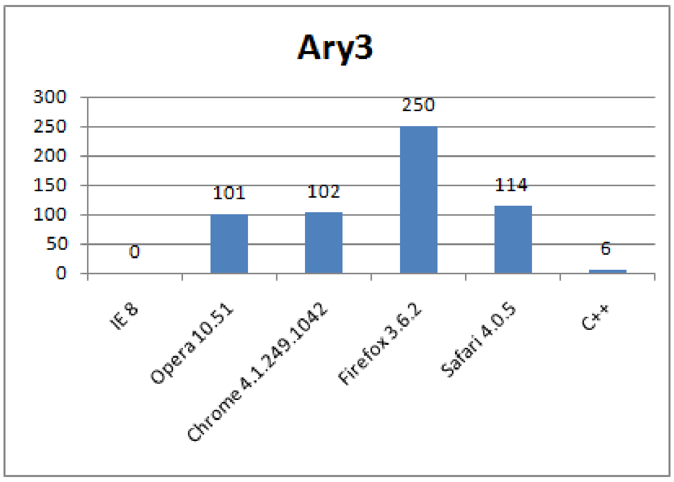


Figure 7.11: Test the speed of accessing arrays

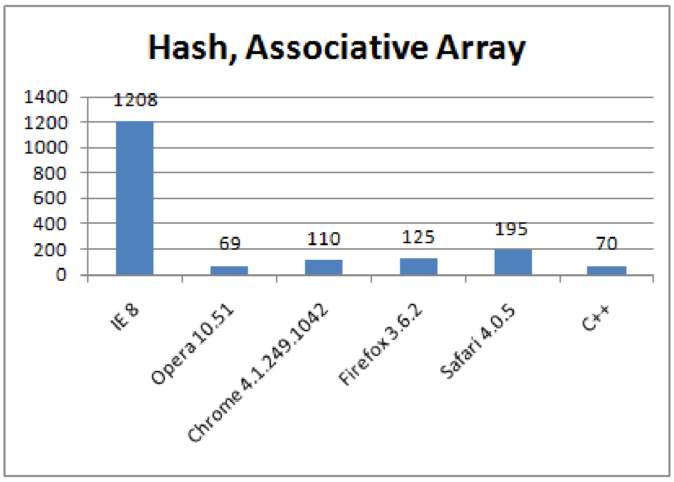


Figure 7.12: Test the speed of reading and writing associative arrays

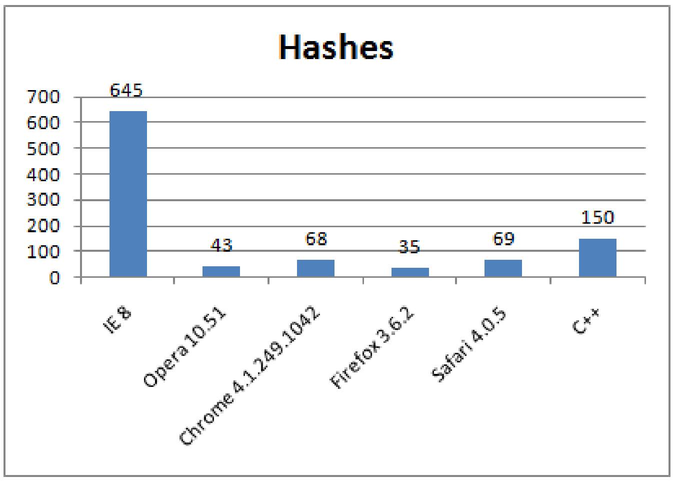


Figure 7.13: An alternative version of the test above, working with two arrays

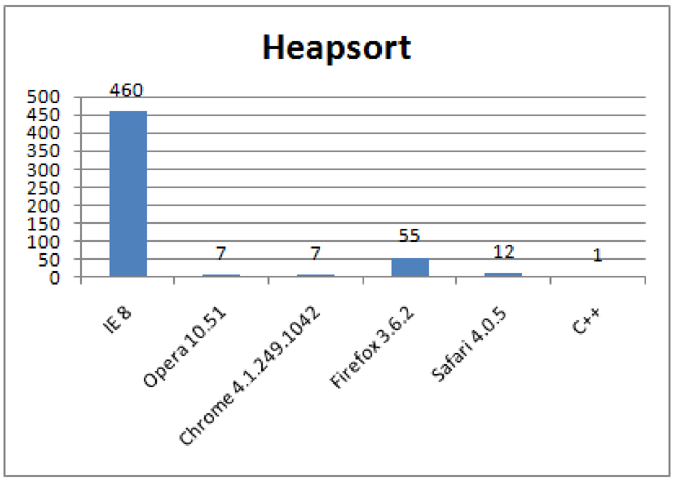


Figure 7.14: Runs a performance test of the comparison-based sorting algorithm heapsort

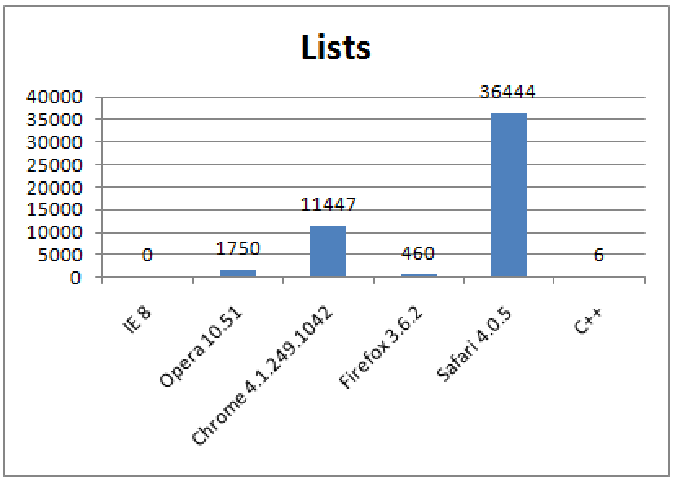


Figure 7.15: Uses arrays to push, concatenate and reverse items. This test performs very different depending on which JavaScript engine it's run in and how that engine chooses to implement the array and its operations.

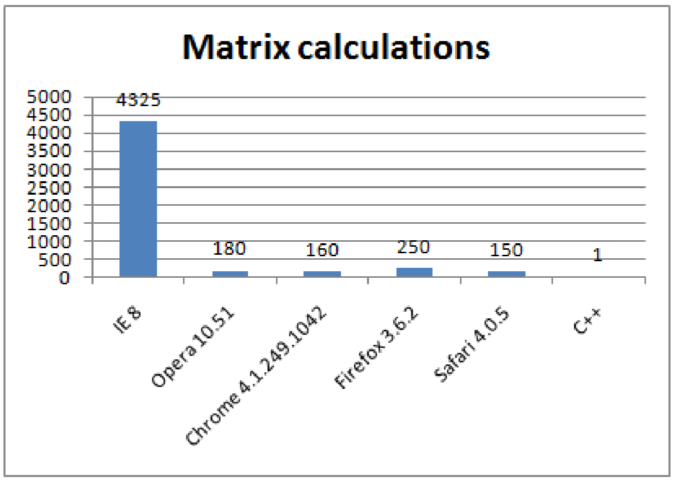


Figure 7.16: Performs matrix multiplications

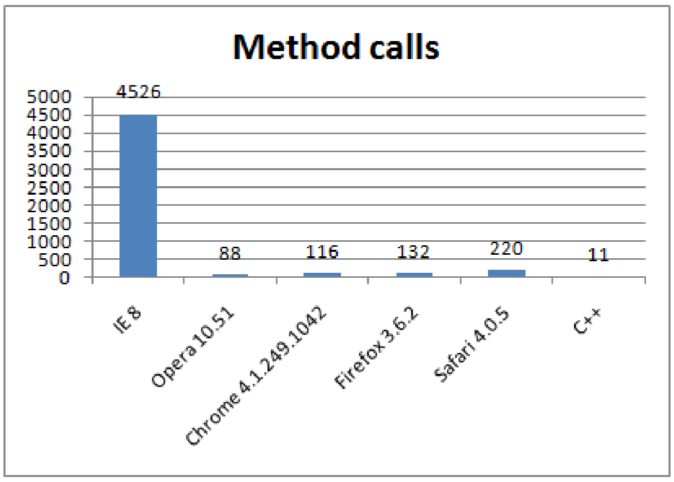


Figure 7.17: Benchmark how fast methods can be called

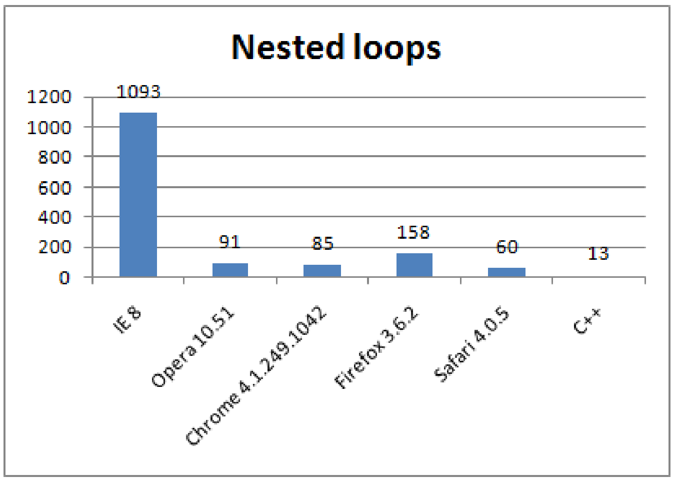


Figure 7.18: Runs a series of deeply nested loops

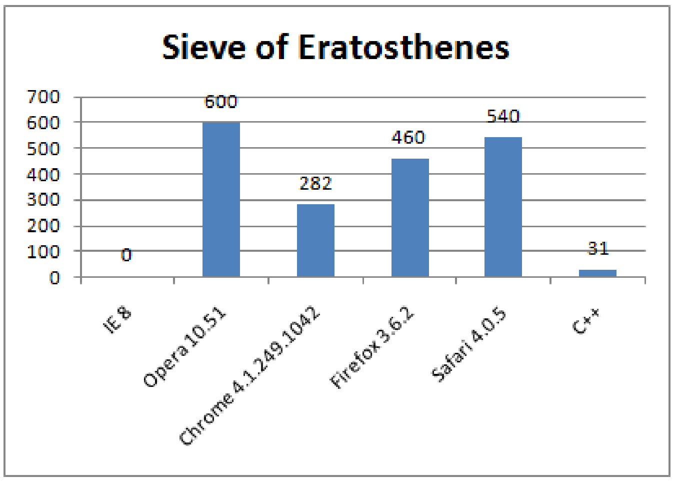


Figure 7.19: The Sieve of Eratosthenes is a simple algorithm for finding all prime numbers up to a specified integer. In this case: 900.

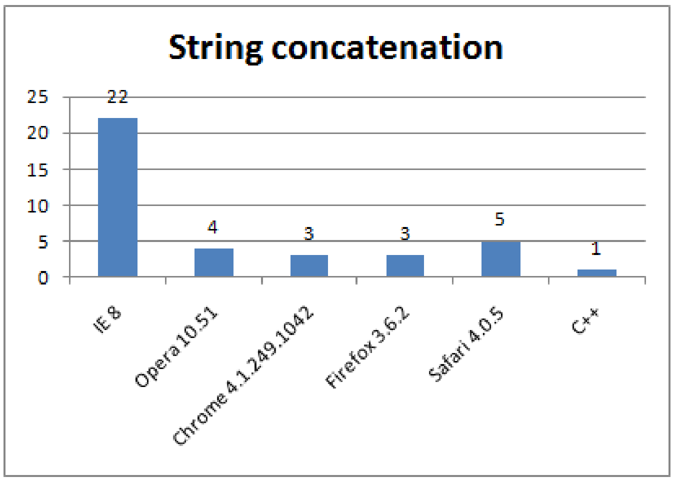
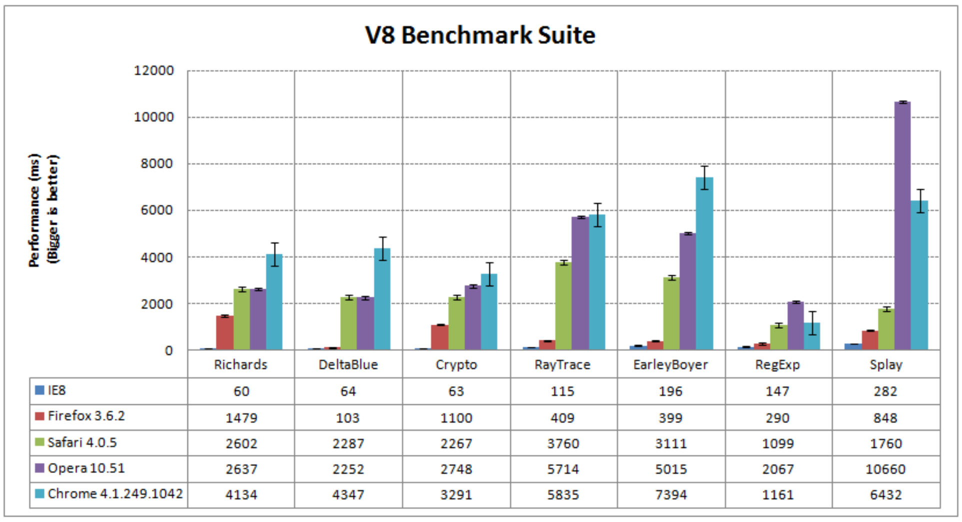


Figure 7.20: Measures the speed of string concatenation

7.7 Test Suites

7.7.1 V8 Test Suite



	Richards	DeltaBlue	Crypto	RayTrace	EarleyBoyer	RegExp	Splay
🟦 IE8	60	64	63	115	196	147	282
🟥 Firefox 3.6.2	1479	103	1100	409	399	290	848
🟩 Safari 4.0.5	2602	2287	2267	3760	3111	1099	1760
🟪 Opera 10.51	2637	2252	2748	5714	5015	2067	10660
🟦 Chrome 4.1.249.1042	4134	4347	3291	5835	7394	1161	6432

Figure 7.21: V8 Test Suite

See appendix H (page 135) for an explanation of what the tests mean.

7.7.2 SunSpider Benchmark

See appendix H (page 132) for an explanation of what the tests mean.

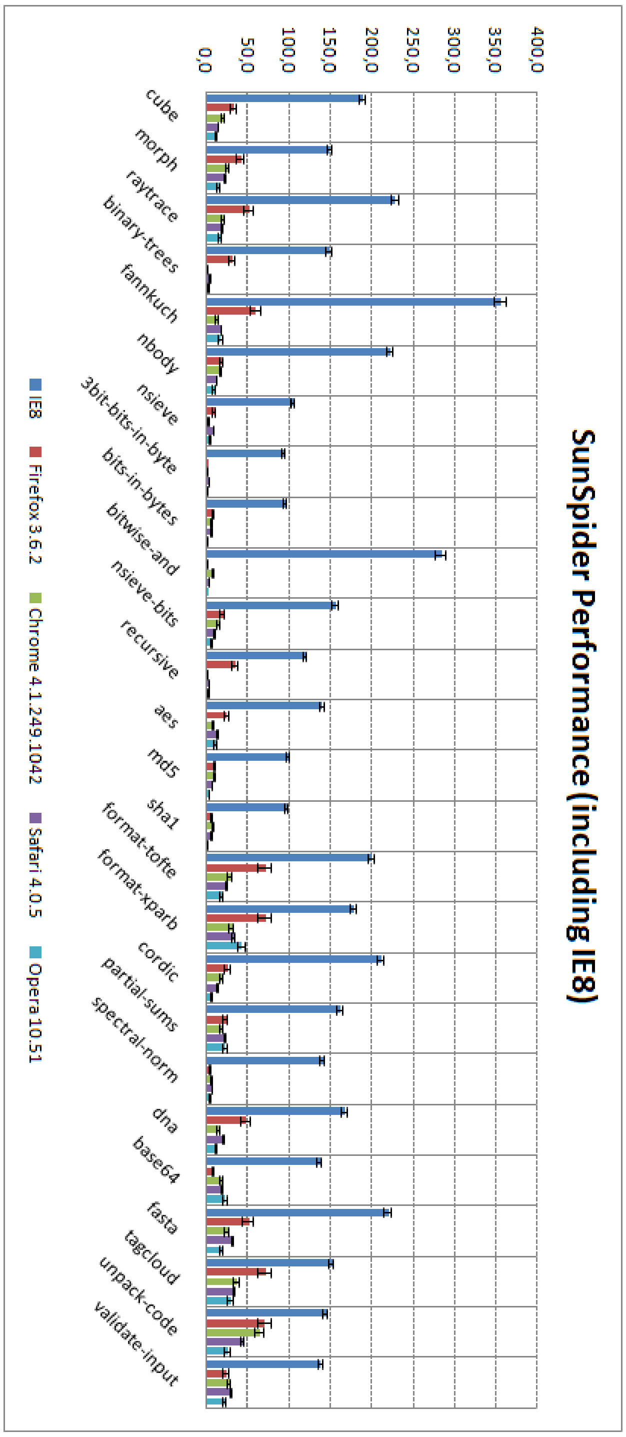


Figure 7.22: SunSpider Performance (including IE8)
Note how slow Internet Explorer is compared to all other competitors, and remember that it has more than 57% of the browser market [23].

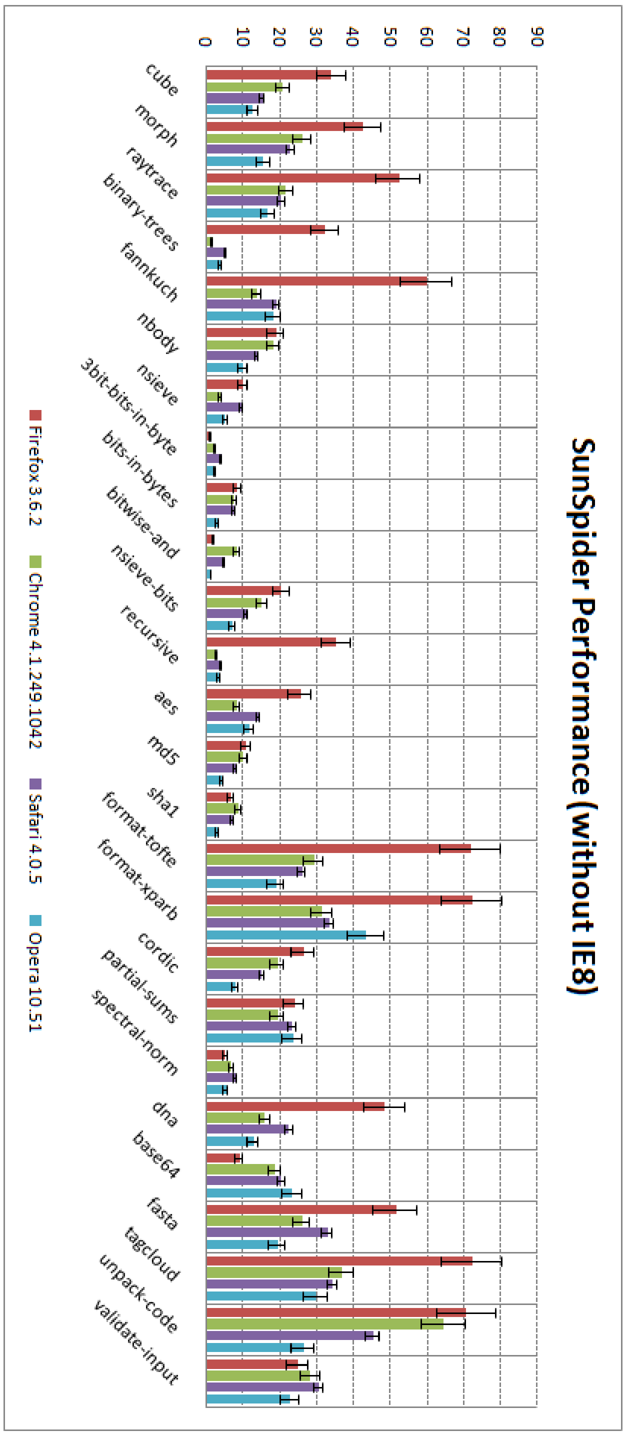


Figure 7.23: SunSpider Performance (without IE8)
Note how slow Firefox is compared to all other competitors, and remember that it together with Internet Explorer have more than 88% of the browser market [23].

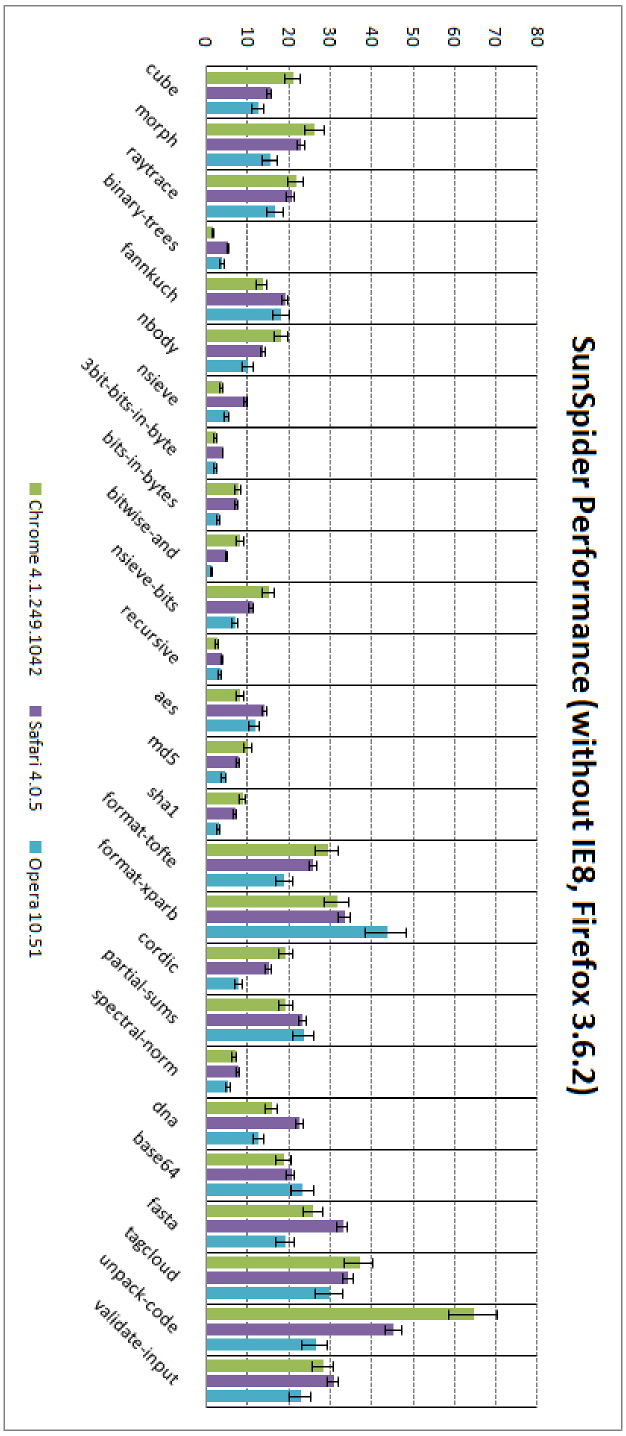


Figure 7.24: SunSpider Performance (without IE8 and Firefox 3.6.2)

Chapter 8

Discussion

8.1 Historic Development

Until 2007, little happened in the terms of performance enhancements in web browsers. Firefox was somewhat faster than it's competitor Internet Explorer, but had little market share. The introduction of the iPhone changed this. Apple rewrote the web performance field by introducing Safari 3. With Safari, they wanted to give their users the same browsing experience on the iPhone, as using a browser on a desktop computer. To be able to do that, they needed to invent new optimizations to both parse, load and render web pages faster while keeping the memory usage to a minimum. Especially optimizations to JavaScript were important, to better handle the new interactive web sites, like Gmail and Facebook, with a behavior similar to desktop applications.

Firefox growing market share and Apple's release of Safari were important, because they started a race among web makers to create faster and faster browsers. With shrinking market share, Microsoft realized they had to act. In the past, they had been totally dominant on the web with Internet Explorer, and had put little effort into complying with industry web standards. The growing amount of Mac owners[1] and many Windows users changing their default browser to Firefox, forced them to change their mind.

In 2008, Apple released a new version of Safari, four times faster than its predecessor. This inspired Google to release their Chrome browser in December the same year, and in 2009 Microsoft finally released a faster and more standards compliant version of their Internet Explorer. Because of the large market share, Microsoft's actions are of great importance. Developers can't write software that's just fast enough to run in some browsers. Standards compliance means that developers just have to deal with one source code, instead of having to write different versions of the same software, depending on in what browser it will be run.

[1]Internet Explorer isn't available for Mac OS X.

In March 2010, Microsoft announced promises of increasing the performance in Internet Explorer, and put more serious effort in complying with standards. If the promise holds, demanding web applications will be adopted by a much broader audience, making the move from the desktop out on to the web even more realistic. What it also could mean is the abandonment of third party non-standard technologies like Flash, Java and Silverlight. Good performing JavaScript engines and web browsers' compliance with standards, like HTML5's video tag, removes the practical value of such technologies.

8.2 Performance of the GIF Decoder and JPEG Encoder

In this thesis, the primary focus has been on the GIF decoder. The reason for this was because GIF decoding and represents tasks traditionally uncommon for scripting languages, namely to do mathematical operations on large amounts of binary data quickly. If a programmer can perform such operations on images quickly enough, it open up doors for creating web based applications like image editors, advanced games and office suite applications. During the implementation of the decoder, I found a similar project, the JPEG encoder which is relatively similar to what I was implementing, so I chose to include it for comparison.

Before starting writing this thesis, I already knew of the existence of some high performance applications. Programmers had already created JavaScript versions of video game emulators[2], 3D games[3] and PowerPoint like applications[4]. I could, however, not find a single scientific paper comparing JavaScript performance in web browsers based on other tests than *Apple's SunSpider* or *Google's V8 Test Suite.* Those suites' test important scenarios, but they're hard to compare to something *real*, like how fast their results are compared to C++. They easily become relative, only comparing their performance development among other browsers. For this reason, I created a GIF decoder. It's something practical that's easy to relate to, as image viewers comes with every modern operating system.

When I implemented the decoder, it was hard to know what performance I could expect from JavaScript, as the research papers I read in connection to this thesis didn't shed any light on the matter. I initially thought that JavaScript would perform similar to Python, Ruby and other popular dynamic languages. At the time, it seemed reasonable to assume that, because dynamic languages have grown in popularity a lot the last years, and their interpreters have went through several speed and memory optimizations,

[2]JSNES, http://benfirshman.com/projects/jsnes

[3]Quake 2 GWT Port, http://code.google.com/p/quake2-gwt-port

[4]Opera Shine, created by me and Rickard Eilert.

and the first two versions that I wrote confirmed this. In *Results*, figure 7.3 shows that version 1 and 2, best run in Chrome, ran almost 10 times slower compared to C++. Running around 10-20 times slower than compiled code is a figure most people would associate with a high level language, because simplicity often means increased productivity at the cost of speed.

I found, after having profiled and optimized the decoder, that it was indeed possible to make it faster. I tried tried two different approaches, using strings and arrays to represent the image data, and version 3, run in Chrome, ran only three times slower than C++, making it the fastest. There's likely room for more improvement here, but not on the current implementation. Before writing version 1-4 I only had a basic understanding of what data structures and design choices that would be best to use to gain maximal performance. I learned that most engines didn't react well to arrays or strings that grew dynamically very often, so I created fixed size arrays which needs resizing very seldom. This required more memory but increased the speed. Ideally, I would have wanted some structure storing binary data efficiently.

The results also show that it's questionable if it's worth the time and lines of code to optimize some parts of the JavaScript by implementing it in the engine. I argue that it's not, and that it's better to put that focus and time into finding tricks making the JavaScript code run faster by itself, without addition of C++.

The decoder was based on a lot of C++ code, with most unnecessary parts removed. Still, the design of the decoder isn't optimal for JavaScript. The whole design idea behind the C++ version was efficient use of data structures and copying larger amounts of data very seldom, and instead create links to the data pieces and put them all together in the end. If a new decoder would be written that uses fixed size arrays or fixed size strings, with a design minimizing the creation of lots of temporary objects or the copying of data back and forth, it might be possible to get the speed seen in the JPEG encoder.

The JPEG encoder results are very interesting because they show that correctly written JS code can give a performance that's only 1.7 times slower than C++. Of course, that's still slower, but in general dynamic languages requires less code. The JavaScript GIF decoder has more than 20% smaller code than it's C++ equivalent, and that source is even based on C++ code to begin with. Other implementations discussed later in this chapter show that the difference of amount of code can be much larger.

The JPEG test also showed another interesting fact: the current implementation of WebWorkers in some browsers, enabling the programmer to take advantage of threads, give almost no performance boost or even worsen the performance. I would guess that's because the current implementations don't run on JavaScript engines that use several threads. For example, Chrome run every open tab in its own thread, but it possible it doesn't allow the JavaScript engine in that tab to create several threads.

Assume that it on average takes a programmer half the time to write something in JavaScript compared to C++. Let's further assume that the code in general runs 2-3 times slower. If the project is large enough, perhaps requiring some months of work time, it's possible that it's more beneficial to buy faster computers rather than switching back to C++. Additionally, the code runs everywhere and don't have to be tested on different architectures, but just in a few browsers.

The downside with using JavaScript is the fact that the code don't run equally fast in every browser. The remarkable speed mentioned above only holds for Chrome, Safari and Opera. That's the browsers with fewest users. Firefox and Internet Explorer together has 88% of the market but run JavaScript 6-24 times slower.

What also concerns me, when looking at the GIF decoder results, is the memory usage. The JavaScript implementation uses 4 times more memory, somewhat lower if the source first is optimized using Google's Closure Compiler[5]. These measurements were performed using only the standalone version of Opera's JavaScript engine, but the same tests could of course be run in the other engines too. The reason for only running it in Opera was because it was least cumbersome to measure. Because of the company's long reputation of producing web browsers consuming least resources of its competitors, I'm confident that I can trust the memory usage figures. The most important thing was to get an idea of how much overhead JavaScript causes, and it's high, probably higher in the others, especially Firefox and IE. Memory might be cheap, but four times is a lot. A mixture of smarter memory management and better written JavaScript code might be the solution.

8.3 XML Parsing

The XML parser surprised me and show something I hadn't expected. Not only is the JS version's code three times smaller, but runs close to or even at the same speed as its C++ equivalent. One could of course argue that the C++ code could be optimized more, that another design could be used etc. However, in reality, many web pages and desktop software are not written by people optimizing them to their fullest extent. It's not reasonable to spend work hours optimizing a product that the customer already finds acceptable. Most applications are made to run just *good enough*.

[5]http://code.google.com/closure/compiler

8.4 Parse 'n Load

This test investigated two important things: how long time a larger script takes to load and if other web sites are responsive during that time. It's very important to have responsiveness even under high load, because a programmer can never control what other web sites scripts do, for how long or stop them. This test showed that there's no correlation between a browser with a slow JavaScript engine and how good it is at cooperative multitasking. In fact, Internet Explorer with its slow JavaScript engine was more responsive than Safari.

8.5 Other Results

I've included elementary tests based on source code from *The Great Win32 Computer Language Shootout* [22]. This was mainly to have something similar to SunSpider and V8 to compare with but also to see how much JavaScript and C++ code it requires to write such test cases. On average, C++ required 50% more code. Interestingly enough, several tests ran at equal or almost equal speed to C++, but many also ran much slower. JavaScript still needs much more optimizations to get up to the same speed of C++ in matrix calculations, array structures, nested loops, lists and method calls.

What also can be seen is how extremely slow IE is compared to all its competitors. This is a huge problem, because the browser has such a large market share which indirectly mean developers either have to lower their ambition to the performance level of IE, ignore the browser and its users completely or implement the product in third party non-standard solutions like Adobe Flash. Hopefully, Microsoft will introduce JIT compilation in version 9 of its browser and correct many of its performance issues.

The SunSpider and V8 benchmark suite results were included for comparison of this thesis to other scientific papers using the same tests. The results show about the same as other papers have found. Internet Explorer lacks behind, Firefox is a bit faster and the fastest browsers are Safari, Chrome and Opera.

8.6 Development Tools

I didn't thoroughly analyze all available tools for web development in this thesis, but a tried a few. I became especially fond of Notepad++, Eclipse, KDevelop and the JavaScript utilities Firebug (Firefox extension), Dragonfly (available in Opera) and Developer Tools (built-in in Chrome). Also, jslint.com proved invaluable to find errors when the browsers JavaScript engines didn't even give an error or finish the parsing of the source code.

Notepad++ gives the programmer syntax high-lightning. Eclipse gives the programmer syntax high-lightning, some basic code hinting and a kind of lint checking. KDevelop gives syntax high-lightning and some code hinting. I tried some other editors and IDEs, but no one felt like *the one*. It would helped a lot if it had existed a single environment with both syntax high-lightning, good lint checking, advanced debugging (like in Dragonfly), profiler (like in Firebug), real code hinting and built-in documentation for the most popular add-on libraries.

8.7 Some Last Thoughts

In the *Related Work* chapter, I wrote that scientists Mikkonen and Taivalsaari saw JavaScript's extreme tolerance for errors as a deficiency. Even if that specific feature in JS is seen as a design flaw, there are methods to work around it. During the work on this thesis, a significant time was spent on porting C++ code to it's JS equivalent. My primary concern was that the ported code would look correct but in reality behave differently. At first, my concern was justified. The syntax of JS and C++ are very similar and because of the non-existent type checking, it's easy to make careless mistakes. These initial mistakes were quickly reduced to a minimum using efficient development tools, like a lint checker, debuggers available in the tested web browsers and by placing assert checks at the beginning of every function.

I implemented an assert function that stopped the program execution when a function or algorithm behaved unexpectedly, by throwing exceptions and printing useful error messages. This assert function was called at the beginning of every function or before important code lines. At times, it was frustrating to write assert statements everywhere, because I was forced to reflect on what the code was supposed to do. Ultimately, that made writing the documentation easier and sometimes made me rewrite code to make it better or more readable. The time spent writing the assert cases was well invested. It gave a feeling of robust code, as errors were easy to pinpoint and solve. Additionally, I added comments before every function statement to explain their purpose, and also where I saw it necessary in the code.

The researchers also criticize JavaScript for its lack of modularity. It's peculiar to express something like that, because it's simply not true. The language is, thanks to its prototype design, different from the Java and C++ family. The reality of JS was perhaps a bit different in 2006-2007, in terms of fewer public add-on libraries, however, not to the magnitude that Mikkonen and Taivalsaari's statements can be considered valid.

JavaScript is extremely dynamic and flexible, and what's missing can easily be added. Either you add the functionality you lack yourself, or use well-established libraries like Dojo or Prototype for that purpose. Naturally, one could argue that adding missing functionality runs the risk of lowering performance and that developers repeatedly reinvent the same solutions independent of each other. Luckily, browsers like Opera and Chrome partially solve this problem by incorporating popular JS libraries inside their JS engines. When a web page requests a popular library it's already byte-code or machine code compiled, decreasing the consumption of both time, memory and CPU usage. The imagined performance losses mentioned in the report is of no concern anymore.

Another concern of the researchers was the lack of functionality, like 2D graphics or networking. At present, all those *defects* have been added or are about to. Most browsers support vector graphics through SVG or Flash, some already support hardware accelerated 2D graphics, while others are planning to implement it, a standard for 3D support is being written, and audio and video support is soon supported in every modern browser, without the need for third party plugins. There even exist working implementations of 3D shooters like Quake 2 written in JavaScript using WebGL. A kind of network support have been added in a few browsers, in the form of TCP based WebSockets, but it's not genuine networking, and has several restrictions due to security reasons. For example, it's important that a client can't be abused as a spam carrier.

Thread management[6] and local database[7] support have recently been or are about to get included in popular web browsers. This might enhance performance for certain types of applications and open up doors for entirely new possibilities of software, once browsers implement the thread management more efficient than today.

When reading the Sun report, one has to keep in mind that it was written about 1-2 years before really fast JavaScript engines were available. In the paper, they criticize JavaScript's prototype design and write that it obviously can be used to implement a software like *the Lively Kernel*, but its prototype nature makes it hard to work with and possibly much slower. Ideally, they would probably have wanted JS to have properties and speed similar to C or C++. It could also be that the scientists are more familiar with C++, and JavaScript whose design philosophy is fundamentally different can take a while to get used to.

Since the paper was written a lot have changed. Since 2002, the popularity of dynamic languages like JavaScript, Python and Ruby has steadily increased, while older traditional languages such as Java, C and C++, has lost many users [24]. JavaScript engines run scripts much faster than back in 2007, there exist several libraries[8] and tools[9] that ease the development of complex programs and the memory consumption is not as severe anymore. Perhaps Mikkonen and Taivalsaari would have reached other conclusions if

[6]WebWorkers, http://www.whatwg.org/specs/web-workers/current-work

[7]Often implementations of SQLite.

[8]jQuery, Scriptalicious and Prototype

[9]Google Web Toolkit

they would have been able run their code in Safari 4, Opera 10.51 or Chrome 4.

Sun's report criticized JS engines for poor memory management, which was true in 2007. Some implementations can still be considered to have this problem, but a majority of the current generation of engines have efficient garbage collectors with clever allocation and quickly release memory when it's no longer being used.

Chapter 9

Conclusion

9.1 Is the Performance of the Software Affected?

Indeed it is, but it depends on the task. If a browser with a really fast JavaScript engine is being used, like Chrome or Opera, one will on average get 2-3 times slower performance compared to C++. Internet Explorer is so slow that it falls into its own category.

9.2 Has the Code Become Easier to Comprehend?

Yes and no. The code might look very similar to C++ but doesn't always behave that way. What is very positive is that the programmer never has to think about memory management, compile the program and that the code in general becomes 50% smaller.

9.3 What Development Utilities Exist and What's Their Impact on the Development?

There are several utilities to aid in the development, but of those I tried no one proved to be a *one solution* development suite. I recommend using the built-in development aids and available extensions in Opera, Firefox and Chrome together with a good editor like Eclipse, Notepad++ or KDevelop. Also, using jslint.com to check the source code is a must. In conclusion, there's much more work needed on Eclipse or similar tools before JavaScript has a development suite as good as Visual Studio.

9.4 When and Where is it Motivated to use JavaScript?

If the user runs any of the three fastest browsers one could possibly use JavaScript to create almost any software. It's not possible, yet, to create high end games like Battlefield, Fallout 3 or Doom 3, but the performance might be good enough to create a less detailed version of World of Warcraft. Because JavaScript resembles C/C++ and C# a lot in it's syntax, porting certain games and software can be quite easy. It's especially interesting that the source code can run everywhere, even on mobile devices.

I would say that it's motivated to use JavaScript to create most common applications today, but with the downside that the performance might be like the one seen on computers 3-4 years old. Implementing a word processor like Word 2007 should be feasible, while a 3D editor like Blender 3D probably wouldn't work, yet.

The language is an open standard and enjoys strong support from the large influential companies and organizations like Google, Apple Inc., Mozilla Foundation och Opera Software.

9.5 How Often is the Code Used, Every Millisecond, Minute, Hour or Day?

Of course, this depends on what one implements, but if it's a GIF decoder, it might be used several times for every page load. A XML parser might be used one time for parsing every new page, or used several times every minute as part of a larger web application like Facebook.

It's important to have this specific question in mind when motivating when and where to use JavaScript as a speed loss might be of less importance if the software is only run once in a while, but of great importance if run every minute.

9.6 Conclusion

JavaScript has over the years gone from being a slow and rather limited language, to today have become feature-rich and fast. It's speed can be around the same or half of comparable code written in C++, but this speed is directly dependent on the choice of the web browser, and the best performance is seen in browsers using JIT compilation techniques.

Even though the language has seen a dramatic increase in performance, there's still major problems regarding memory usage. JavaScript applications typically consume 3-4 times more memory than similar applications written in C++. Many browser vendors, like Opera Software, acknowledge this and are currently trying to optimize their memory usage. This issue is hopefully non-existent within a near future.

Chapter 10

Future Work

This report hasn't measured real web sites, like the reports from Microsoft. Creating an application that can be used to record the behavior of a user's visit to web sites and easy let them be replayed in all major web browsers, would possibly help a lot in finding new areas to optimize. Such tool would also eliminate some uncertainty like network lag and response times from web servers or databases, as all replay-information are stored locally. Such utility could not be written directly using JavaScript because of the JS-implementations cross-domain restrictions. Instead, I would recommend looking into creating such utility as an extension or plugin.

There are utilities like Reducio[1] that can reduce the code paths taken in JavaScript to an absolute minimum. Opera uses this utility to quickly track the source of bugs. It could ideally be used to optimize web sites and their libraries, to only contain the code that's absolutely needed and remove all cold paths. In reality, it's almost never feasible to do this automatically. The code paths might depend on user input or be browser specific, and before there exist certain methods to make sure the code is meant for a specific browser or will never be called, it can't be removed. Another interesting approach would be to create heuristics that ignore libraries until they're called. Creating such algorithms are difficult since we don't know what the libraries contain. For example, they might change global settings affecting the functionality of other libraries. An approach that's worth investigating is what the caching system can remember from the first parsing of a page. The parser might learn that two libraries are strictly needed, while a third can be lazily loaded later, as its functions seldom are called.

Researchers at the Korea Advanced Institute of Science & Technology have tried an approach where they simplified JavaScript and made the language syntax less complex to speed it up. They were inspired by a paper written by Thiemann[2], in which he proposed a type system for JavaScript.

[1]Developed internally at Opera Software by Geoffrey Sneddon and James Graham.

[2]Professor Peter Thiemann of Universität Freiburg.

The type system is mainly intended as a way to analyze JavaScript code before deployment, a sort of lint-checker. The Korean approach seems successful of obvious reasons, but with the catch that it removes the dynamics of the prototype language. [25, 8]

Chapter 11

Appendix A

Inspired by Björn Lindqvist's decompressor.
http://bjourne.blogspot.com/2007/11/example-of-lzw-algorithm.html.

11.1 LZW decompressor written in Python

Listing 11.1: LZW decompressor

```
def decompress(lzw_stream):
    # LZW table that maps codes to strings
    table = [chr(i) for i in range(256)]
    # The first code is always added to output
    prevcode = lzw_stream[0]
    output = []
    output.append(table[prevcode])

    for code in lzw_stream[1:]:
        try:
            # Is the code already defined?
            entry = table[code]
        except IndexError:
            # The code is not yet defined
            entry = table[prevcode]
            entry += entry[0]
        output.append(entry)
        # Lookups the previous code and combines it
        # with the current entry
        table.append(table[prevcode] + entry[0])
        prevcode = code
    return output

data = open("lzwdata.txt").read()
print decompress(data) # Print the decompressed data to stdout
```

Chapter 12

Appendix B

12.1 GIF Files Used During Implementation

12.1.1 GifSample

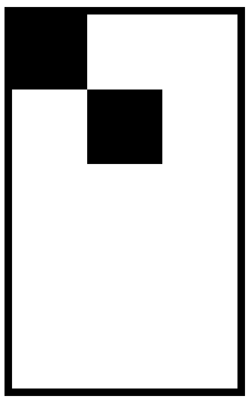


Figure 12.1: GifSample

Size: 32x52 px
http://en.wikipedia.org/wiki/File:GifSample.gif

12.1.2 OpenDNS Logo

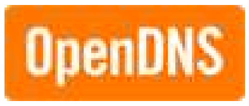


Figure 12.2: OpenDNS Logo

Size: 100x40 px
http://www-files.opendns.com/img/home-footer-logo.png

12.1.3 Colored Small Test Image

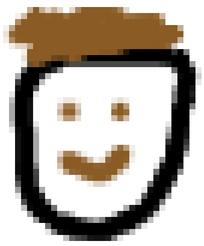


Figure 12.3: Colored Small Test Image

Size: 40x40 px
Created during the implementation of the decoder.

12.1.4 i-Bench Transparent GIF

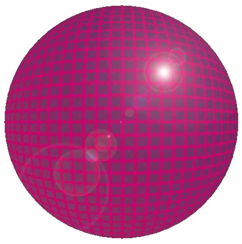


Figure 12.4: i-Bench Test Suite, Transparent GIF

Size: 432x432 px
From the i-Bench 5.0 Test Suite. [26]

12.1.5 Large Colored Sinus Curve

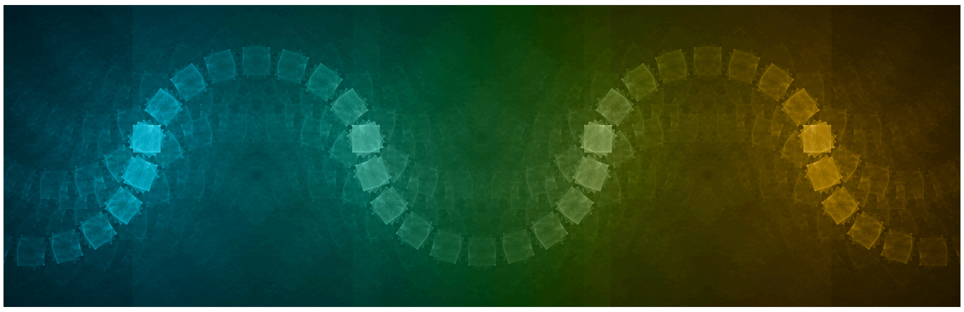


Figure 12.5: Large Colored Sinus Curve

Size: 3340x1050 px
Created by Opera employee Tim Johansson.

12.2 Strange Results when Decoding GIF

12.2.1 Missing Color Channels

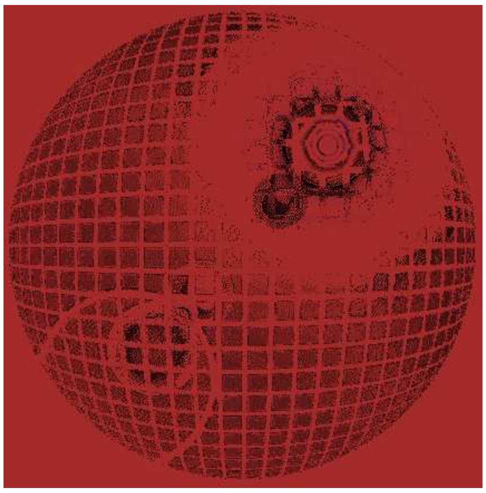


Figure 12.6: A bug in the JavaScript decoder caused it to drop various pixels and to only decode the red channel

Size: 430x433 px
Created during implementation of the GIF decoder.

12.3 PNG File Used with the JPEG Encoder

12.3.1 Pool table

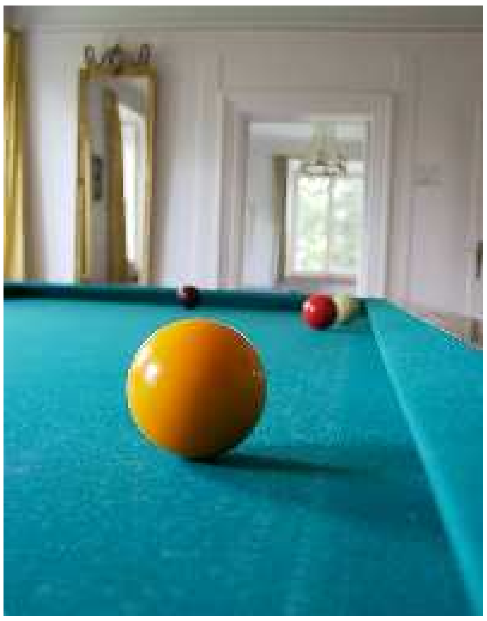


Figure 12.7: The image used to benchmark the JPEG encoder

Size: 200x255 px
Created by Andreas Ritter [5].

Chapter 13

Appendix C GIF Format Explanation

13.1 Defining the Grammar

Symbols	Purpose
<>	Grammar word
::=	Defines symbol
$*$	Zero or more occurrences
+	One or more occurrences
|	Alternative element
[ ]	Optional element

Table 13.1: Grammar symbols

13.1.1 Grammar

<GIF Data Stream> ::= Header <Logical Screen> <Data>* Trailer

<Logical Screen> ::= Logical Screen Descriptor [Global Color Table]

<Data> ::= <Graphic Block> | <Special-Purpose Block>

<Graphic Block> ::= [Graphic Control Extension] <Graphic-Rendering Block>

<Graphic-Rendering Block> ::= <Table-Based Image> | Plain Text Extension

<Table-Based Image> ::= Image Descriptor [Local Color Table] Image Data

<Special-Purpose Block> ::= Application Extension | Comment Extension

13.1.2 Example

<GIF Data Stream> ::= Header <Logical Screen> <Data> $*$ Trailer

13.2 GIF Data Format

This is the expected order of data in a GIF file, from the beginning to the end.

13.2.1 Header

Name	Bytes	Purpose
Signature	3	Always ”GIF”
Version	3	”87a” or ”89a”

Table 13.2: GIF: Header

13.2.2 Logical Screen Descriptor

Name	Bytes	Purpose
Logical Screen Width	2	Image width
Logical Screen Height	2	Image height
Packed Fields	1	See the table below.
Background Color Index	1	Index into the Global Color Table for the Background Color. The Background Color is the color used for those pixels on the screen that are not covered by an image. If the Global Color Table Flag is set to (zero), this field should be zero and should be ignored.
Pixel Aspect Ratio	1	Gives the Pixel Aspect Ratio. Has to be calculated manually if the field is set to zero. ((width/height)+15)/64. If the field has a value, calculate the aspect ratio using $(\text{value} + 15)/64$.

Table 13.3: GIF: Logical Screen Descriptor

Name	Bits	Purpose
Global Color Table Flag	1	If set, has a Global Color Table. If not set, the value of Background Color Index below is ignored.
Color Resolution	3	Number of bits per primary color available to the original image, minus 1. This value represents the size of the entire palette from which the colors in the graphic were selected, not the number of colors actually used in the graphic. For example, if the value in this field is 3, then the palette of the original image had 4 bits per primary color available to create the image. This value should be set to indicate the richness of the original palette, even if not every color from the whole palette is available on the source machine.
Sort Flag	1	If the flag is set, the Global Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists the decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic. A value of zero means *not ordered*, while one means *ordered by decreasing importance, most important color first.*
Size of Global Color Table	3	If the Global Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Global Color Table. To determine that actual size of the color table, $2^{\text{the value of the field} + 1}$. Even if there is no Global Color Table specified, set this field according to the given formula so that decoders can choose the best graphics mode to display the stream in.

Table 13.4: GIF: Logical Screen Descriptor, Packed Fields

13.2.3 Global Color Table

Name	Bytes	Purpose
RGB-triplet	*	A single RGB-triplet is three bytes. The total size of this block is $3 * 2^{(\text{Size of Global Color Table + 1})}$
		This post only exists if the Global Color Table-flag is set to 1.

Table 13.5: GIF: Global Color Table

13.2.4 Image Descriptor

Name	Bytes	Purpose
Image Separator	1	Identifies the beginning of an Image Descriptor. Is always set to 0x2C.
Image Left Position	2	Column number, in pixels, of the left edge of the image, with respect to the left edge of the Logical Screen. Leftmost column of the Logical Screen is 0.
Image Top Position	2	Row number, in pixels, of the top edge of the image with respect to the top edge of the Logical Screen. Top row of the Logical Screen is 0.
Image Width	2	Width of the image in pixels.
Image Height	2	Height of the image in pixels.
Packed Fields	1	See the table below.

Table 13.6: GIF: Image Descriptor

Name	Bits	Purpose
Local Color Table Flag	1	Uses the Global Color Table, if set to 0.
Interlace Flag	1	The image uses interlacing if this value is set to 1. Interlacing is handled using a ”four-pass interlace pattern”.
Sort Flag	1	If the flag is set, the Local Color Table is sorted, in order of decreasing importance. Typically, the order would be decreasing frequency, with most frequent color first. This assists a decoder, with fewer available colors, in choosing the best subset of colors; the decoder may use an initial segment of the table to render the graphic.
Reserved	2	Not used.
Size of Local Color Table	3	If the Local Color Table Flag is set to 1, the value in this field is used to calculate the number of bytes contained in the Local Color Table. This value should be 0 if there is no Local Color Table specified.

Table 13.7: GIF: Image Descriptor, Packed Fields

13.2.5 Local Color Table

Name	Bytes	Purpose
RGB-triplet	*	A single RGB-triplet is three bytes. The total size of this block is $3 * 2^{(\text{Size of Local Color Table } + \text{ } 1)}$
		This post only exists if the Local Color Table-flag is set to 1.

Table 13.8: GIF: Local Color Table

13.2.6 Table Based Image Data

Name	Bytes	Purpose
LZW Minimum Code Size	1	The fewest number of bits an LZW code used. For example, 5 bits.
Block Size	1	Size of the Image Data block. At most 255 bytes.
Image Data	*	One or more bytes.
Block Terminator	1	Marks the end of the block. Always set to 0x00.

Table 13.9: GIF: Table Based Image Data

13.2.7 Graphic Control Extension

Name	Bytes	Purpose
Extension Introducer	1	Marks the beginning of an extension. Always set to 0x21.
Extension Label	1	Identifies the block as Graphic Control Extension. Always set to 0xF9.
Block Size	1	Size of the Graphic Control Extension block. Always set to 4.
Packed Fields	1	See the table below.
Delay Time	2	Tells the decoder how many 1/100 of a second it should wait before decoding this particular frame. Used for animations, but ignored if set to 0.
Transparent Color Index	1	Gives the index of the color in the Local or Global Color Table that should be transparent.
Block Terminator	1	Marks the end of the block. Always set to 0x00.

Table 13.10: GIF: Graphical Control Extension

Name	Bits	Purpose
Reserved	3	Ignored.
Disposal Method	3	Indicates how the graphic should be handled after being rendered. See the table below.
User Input Flag	1	Indicates whether or not user input is expected before continuing. If the flag is set, processing will continue when user input is entered. The nature of the user input is determined by the application, like *Carriage Return*, *Mouse Button*, *Click* etc.
Transparent Color Flag	1	Is set if a transparent color is being used.

Table 13.11: GIF: Graphical Control Extension, Packed Fields

Value	Disposal Method
0	No disposal specified. The decoder shall not take any action.
1	Do not dispose. The graphic is to be left in place.
2	Restore to background color. The area used by the graphic must be restored to the background color.
3	Restore to previous. The decoder is required to restore the area overwritten by the graphic with what was there prior to rendering the graphic.
4-7	Undefined.

Table 13.12: GIF: Graphical Control Extension, Packed Fields, Disposal Method

13.2.8 Comment Extension

Name	Bytes	Purpose
Extension Introducer	1	Marks the beginning of an extension.
		Always set to 0x21.
Extension Label	1	Identifies the block as Comment Extension. Always set to 0xFE.
Comment Data	*	This field can be used to store comments about the image. The first byte give the size of the block, 1–255 bytes. Comments are given in 7-bit ASCII.
Block Terminator	1	Marks the end of the block.
		Always set to 0x00.

Table 13.13: GIF: Comment Extension

13.2.9 Plain Text Extension

Officially rejected since October 1998. An extension with its label set to 0x01 should always be ignored.

Name	Bytes	Purpose
Extension Introducer	1	Marks the beginning of an extension.
		Always set to 0x21.
Extension Label	1	Identifies the block as Plain Text Extension. Always set to 0x01.
Plain Text	*	Plain Text-data. 1 byte or larger.
Block Terminator	1	Marks the end of the block.
		Always set to 0x00.

Table 13.14: GIF: Plain Text Extension

13.2.10 Application Extension

Name	Bytes	Purpose
Extension Introducer	1	Marks the beginning of an extension. Always set to 0x21.
Extension Label	1	Identifies the block as Application Extension. Always set to 0xFF.
Block Size	1	Size of the Application Extension block. Always set to 11.
Application Identifier	8	Eight ASCII characters that identifies what application that uses this extension.
Application Authentication Code	3	Three bytes to authenticate the application that uses this extension.
Application Data	*	Application specific data. 1 byte or larger.
Block Terminator	1	Marks the end of the block. Always set to 0x00.

Table 13.15: GIF: Application Extension

13.2.11 Trailer

Name	Bytes	Purpose
GIF Trailer	1	Marks the end of the GIF file. Always set to 0x3B.

Table 13.16: GIF: Trailer

Chapter 14

Appendix D
A GIF Decoder written in C++

This was the second GIF decoder I wrote. The first was a failure and has not been included. This decoder was not used when collecting benchmark data, and has only been included to show what a GIF decoder can look like.

14.1 main.c

Listing 14.1: main.c

```
#include "gifdecoder.h"

// Converts 1 byte char to a 8 byte char showing the char's
// binary representation.
void binary(unsigned char data, unsigned char* packed_fields) {
   int i;
   for (i = 0; i < 9; i++) {
      if ((data & 0x80) == 0)
         packed_fields[i] = '0';
      else
         packed_fields[i] = '1';
      data = data << 1;
   }
}

int main(int argc, char* argv[]) {
   FILE *file;
   struct gif_t gif;
   unsigned char packed_fields[8];
   unsigned char identity;
   int descriptor = 0, i;
   void *current;
   void *next;

   printf("File: %s\n-------\n", "GifSample.gif");
   file = fopen("GifSample.gif", "rb");
   if (!file)
   {
      printf("Error! Couldn't open GifSample.gif!");
      return 1;
   }

   // Reads the GIF header and Logical Screen Descriptor.
   fread(&gif.header, 6, 1, file);
   fread(&gif.logical_screen_descriptor, 7, 1, file);

   // Prints the header.
   printf("Header\n");
   printf("  Signature: %.3s\n  Version: %.3s\n",
      gif.header.signature,
      gif.header.version);

   // Prints the Logical Screen Descriptor.
   printf("Logical Screen Descriptor\n");
   printf("  Width: %i\n",
      gif.logical_screen_descriptor.logical_screen_width);
   printf("  Height: %i\n",
      gif.logical_screen_descriptor.logical_screen_height);
   binary(gif.logical_screen_descriptor.packed_fields,
      (unsigned char *)&packed_fields);
   printf("  Packed fields: %.8s\n", packed_fields);
   printf("  Background Color Index: %i\n",
      gif.logical_screen_descriptor.background_color_index);
   printf("  Pixel Aspect Ratio: %i\n",
      gif.logical_screen_descriptor.pixel_aspect_ratio);

   if ((gif.logical_screen_descriptor.packed_fields & 0x80) ==
       GLOBAL_COLOR_TABLE_FLAG) {
      printf("Global Color Table ");
      gif.global_color_table = gif_decode_ct(file,
         (gif.logical_screen_descriptor.packed_fields & 0x08)
             ==
         SORT_FLAG_G,
         (gif.logical_screen_descriptor.packed_fields & 0x07));

      // Prints the Global Color Table.
      printf("  Number of colors: %i\n", 1 <<
         ((gif.logical_screen_descriptor.packed_fields & 0x07)
             +
          1));
      for (i = 0; i < (1 <<
           ((gif.logical_screen_descriptor.packed_fields & 0x07
               ) +
            1));
            i++)
         printf("  r: %i, g: %i, b: %i\n",
               gif.global_color_table.color_table[i].r,
               gif.global_color_table.color_table[i].g,
               gif.global_color_table.color_table[i].b);
      printf("[decoded]\n");
   }

   // Initializes the first descriptor.
   gif.image_descriptors = (struct descriptor_t *)gif_malloc(
       sizeof(
      struct image_descriptor_t));
   gif.image_descriptors->descriptor = NULL;
   gif.image_descriptors->next = NULL;
   // Initializes the extensions to NULL.
   gif.extensions = NULL;

   while (gif_next(file, &identity) != 0) {
      switch (identity) {
         case IMAGE_SEPARATOR:
            printf("Image Descriptor ");
            gif_decode_image_descriptor(file, &gif);
            descriptor++;
            printf("[decoded]\n");
            printf("LZW data ");
            gif_decode_lzw(file, &gif, descriptor);
            printf("[decoded]\n");
            break;
         case EXTENSION:
            gif_decode_extension(file, &gif);
            break;
         case END:
            printf("End of GIF! [3B]");
            break;
      }
   }

   // Clean up, and closes the file.
   if (gif.image_descriptors != NULL) {
      current = gif.image_descriptors;
      while (current != NULL) {
         next = ((struct descriptor_t *)current)->next;
         free(current);
         current = next;
      }
   }
   if (gif.extensions != NULL) {
      current = gif.extensions;
      while (current != NULL) {
         next = ((struct extension_t *)current)->next;
         free(current);
         current = next;
      }
   }

   free(gif.global_color_table.color_table);
```

```
    fclose(file);
}
```

14.2 gifdecoder.c

Listing 14.2: gifdecoder.c

```
#include "gifdecoder.h"

class Colors
{
   unsigned int size;
   unsigned int max;
   color_t *colors;
public:
   Colors()
   {
      size = 0;
      max = 12;
      colors = (color_t *)gif_malloc(sizeof(color_t) * max);
   }
   ~Colors()
   {
      free(colors);
   }

   void Add(color_t color)
   {
      if (size == max)
      {
         max += 12;
         colors = (color_t *)realloc(colors, sizeof(color_t) *
             max);
      }

      colors[size].r = color.r;
      colors[size].g = color.g;
      colors[size].b = color.b;
      size++;
   }

   unsigned int GetSize()
   {
      return size;
   }

   color_t GetSingle(unsigned int item)
   {
      return colors[item];
   }

   color_t * GetAll()
   {
      return colors;
   }
};
```

```
class Table
{
   unsigned int index;
   unsigned int size;
   unsigned int color_table_size;
   unsigned int start;
   Colors colors[8192];
   unsigned int old_code;
   Colors *output1;
   Colors *output2;
   unsigned int counter;
   bool decoding;

   void Increase()
   {
      index++;
      size++;
   }

   void UpdateOldCode(unsigned int code)
   {
      old_code = code;
   }

   void SetOutput(Colors *ptr1, Colors *ptr2)
   {
      output1 = ptr1;
      output2 = ptr2;
   }

   void Insert(Colors *value, Colors *parent)
   {
      for (unsigned int i = start; i < size; i++)
      {
         if (colors[i].GetSize() == value->GetSize() + 1)
         {
            if (memcmp(colors[i].GetAll(), value->GetAll(),
                    sizeof(Colors) * value->GetSize()) == 0 &&
                colors[i].GetSingle(value->GetSize()).r ==
                    parent->GetSingle(0).r &&
                colors[i].GetSingle(value->GetSize()).g ==
                    parent->GetSingle(0).g &&
                colors[i].GetSingle(value->GetSize()).b ==
                    parent->GetSingle(0).b)
            {
               SetOutput(&colors[i], NULL);
               return;
            }
         }
      }

      for (unsigned int i = 0; i < value->GetSize(); i++)
         colors[index].Add(value->GetSingle(i));
      colors[index].Add(parent->GetSingle(0));

      Increase();
   }
public:
   Table(unsigned int start, color_table_t *color_table)
   {
      this->start = start;
      color_table_size = color_table->size;
      counter = 0;
      Clear();

      for (unsigned int i = 0; i < color_table->size; i++)
         colors[i].Add(color_table->color_table[i]);
   }

   void Clear()
   {
      index = start;
      // +2 because of clear and end code.
      size = color_table_size + 2;
      decoding = false;
      SetOutput(NULL, NULL);
   }

   bool CodeExist(unsigned int code)
   {
      return (code < this->index);
   }

   void Output()
   {
      for (unsigned int i = 0; i < output1->GetSize(); i++)
      {
         if (output1->GetSingle(i).r == 0 &&
             output1->GetSingle(i).g == 0 &&
             output1->GetSingle(i).b == 0)
         {
             printf("*");
             counter++;
         }
         else
         {
            printf(" ");
            counter++;
         }

         if (counter == 32)
         {
            printf("\n");
            counter = 0;
         }
      }
      if (output2 != NULL)
      {
         if (output2->GetSingle(0).r == 0 &&
             output2->GetSingle(0).g == 0 &&
             output2->GetSingle(0).b == 0)
         {
            printf("*");
            counter++;
         }
         else
         {
            printf(" ");
            counter++;
         }

         if (counter == 32)
         {
            printf("\n");
            counter = 0;
         }
      }
   }

   void Add(unsigned int code)
   {
      Colors *parent;

      if (decoding == false)
      {
         decoding = true;
         UpdateOldCode(code);
         SetOutput(&colors[code], NULL);
         Output();
         return;
      }

      // If a part of a word contains the same code it would
      // have if added to the dictionary, it means it points
      // to "whatever comes next" (itself). A fix for this
      // mystery is to see it as a point to the beginning of
      // itself. The code that comes next is therefore word[0].
      if (CodeExist(code))
      {
         parent = &colors[code];
         SetOutput(&colors[code], NULL);
      }
      else
      {
         parent = &colors[old_code];
         SetOutput(&colors[old_code], &colors[old_code]);
      }

      Insert(&colors[old_code], parent);
      UpdateOldCode(code);
```

```
      Output();
   }
};

// Decodes a color table.
struct color_table_t gif_decode_ct(FILE *file,
   unsigned char sort_flag, unsigned int size)
{
   struct color_table_t table;
   unsigned int i;
   unsigned int total_size = 1 << (size + 1); // 2^(size + 1)
   unsigned int color = 0;

   table.color_table = (struct color_t *)gif_malloc(total_size
       *
      sizeof(struct color_t));
   for (i = 0; i < total_size; i++)
   {
      fread(&color, 3, 1, file);
      table.color_table[i].r = (color & 0x000000FF);
      table.color_table[i].g = (color & 0x0000FF00) >> 8;
      table.color_table[i].b = (color & 0x00FF0000) >> 16;
   }
   table.sort_flag = sort_flag;
   table.size = total_size;

   printf("(%i) ", total_size);
   return table;
}

// Decodes the Image Descriptor.
void gif_decode_image_descriptor(FILE *file, struct gif_t *gif)
{
   struct image_descriptor_t *image_descriptor =
      (struct image_descriptor_t *) gif_malloc(
         sizeof(struct image_descriptor_t));
   struct orientation_t orientation;
   unsigned char data;
   struct descriptor_t *current;

   fread(&orientation, 8, 1, file);
   image_descriptor->orientation = orientation;
   fread(&data, 1, 1, file);
   image_descriptor->interlace_flag = ((data & 0x40) ==
      INTERLACE_FLAG);

   if ((data & 0x80) == LOCAL_COLOR_TABLE_FLAG)
      image_descriptor->local_color_table = gif_decode_ct(file,
         ((data & 0x20) == SORT_FLAG_L),
         (data & 0x07));
   else
      image_descriptor->local_color_table.color_table = NULL;

   // Locates where to store the descriptor and stores it.
   current = gif->image_descriptors;
   while (current->next != NULL)
      current = current->next;
   current->descriptor = image_descriptor;
   current->next = (struct descriptor_t *)gif_malloc(
      sizeof(struct descriptor_t));
   current->next->next = NULL;
}

// Decodes the LZW data.
void gif_decode_lzw(FILE *file, struct gif_t *gif,
   unsigned int descriptor)
{
   // LZW specific data.
   struct lzw_help_decompress_t decompress_data;
   unsigned char *compressed_data;
   unsigned int position = 0;
   unsigned char block_size;
   struct lzw_word_table_t lzw_word_table;
   // Helpers
   struct descriptor_t *current;

   // Read and store the clear, end and start words.
   fread(&lzw_word_table.code_size, 1, 1, file);
   fread(&block_size, 1, 1, file);
   lzw_word_table.clear_word = 1 << lzw_word_table.code_size;
   lzw_word_table.end_word = lzw_word_table.clear_word + 1;
   lzw_word_table.start_word = lzw_word_table.clear_word + 2;
   lzw_word_table.number_of_words = lzw_word_table.start_word;

   // Allocates memory and reads all compressed data.
   compressed_data = (unsigned char *)gif_malloc(block_size);
   fread(compressed_data, block_size, 1, file);

   // Locates the descriptor.
   current = gif->image_descriptors;
   while (--descriptor != 0)
      current = current->next;
   decompress_data.descriptor =
      (struct image_descriptor_t *)current->descriptor;

   // Assigns the correct color table to be used.
   if (decompress_data.descriptor->local_color_table.
       color_table ==
          NULL)
      decompress_data.color_table = &gif->global_color_table;
   else
      decompress_data.color_table =
         &decompress_data.descriptor->local_color_table;

   // Reads and decodes the data.
   printf("\nMinimum code size: %i\n", lzw_word_table.code_size
       );
   printf("Block size: %i\n", block_size);
   lzw_decode(&lzw_word_table, block_size, compressed_data,
      decompress_data.color_table);

   // Free the memory.
   free(compressed_data);

   // Skip the block terminator.
   fseek(file, 1, SEEK_CUR);
}

void lzw_decode(struct lzw_word_table_t *table,
   unsigned char block_size, unsigned char *data,
   struct color_table_t *color_table)
{
   struct lzw_code_t code;
   struct lzw_code_t head;
   struct lzw_code_t tail;
   struct lzw_code_t word[1024];
   unsigned short length = 0;
   unsigned int position = 0;
   unsigned short int padding = 0;
   struct color_t color;
   struct print_queue_t print_queue;
   // [0 to (table->clear_code - 1)] are indexes in the color
       table
   // (LCT/GCT).
   unsigned int index = table->start_word;
   unsigned short code_size =
      lzw_get_bit_size(table->code_size, table->clear_word);
   unsigned short int needed = code_size;
   unsigned int i;
   unsigned int j;
   unsigned short first = 1;
   unsigned int output;
   unsigned short add = 0;
   unsigned char special = 0;
   struct remove_t remove;
   remove.length = 0;
   remove.remove = 0;
   Table table1(table->start_word, color_table);

   // Retrieves and decodes the words.
   code.code = 0;
   code.bits = 0;
   head.bits = 0;
   tail.bits = 0;
   print_queue.length = 0;
   while (position < block_size)
   {
      // Check if there are remaining bits from previous
          iteration.
      if (tail.bits > 0)
      {
         head.code = tail.code;
         head.bits = tail.bits;
         tail.bits = 0;
      }

      // If no data in buffer, read data.
      if (head.bits == 0)
      {
         head.code = (unsigned char)data[position++];
         head.bits = 8;
      }

      // If the bits we need are fewer than the available.
      if (needed > head.bits)
      {
         code.code += head.code << padding;
         code.bits += head.bits;
         needed -= head.bits;
         padding += head.bits;
         head.bits = 0;
      }
      else // All bits we need are available.
      {
         // Puts the whole code together
         code.code += (BITGET(head.code, needed) << padding);
         code.bits += needed;

         // Save any remaining bits.
         tail.bits = 8 - needed;
         tail.code = BITGET(head.code >> needed, tail.bits);

         if (code.code == table->clear_word) // Clear word
             found
         {
            printf("--> Clear word found!<--\n");
            code_size = lzw_get_bit_size(table->code_size,
               table->clear_word);
            needed = code_size;
            table1.Clear();
            length = 0;
            code.code = 0;
            code.bits = 0;
            head.bits = 0;
            padding = 0;
            continue;
         }
         else if (code.code == table->end_word) // End code
             found
         {
            printf("--> End word found! <--\n");
            break;
         }

         table1.Add(code.code);
         code.code = 0;
```

```
         code.bits = 0;
         head.bits = 0;
         padding = 0;
         needed = code_size;
      }
   }
}

struct color_t get_color(unsigned int code,
   struct color_t *color_table)
{
   if (code > 255)
      printf("OMG! (%i)\n", code);
   return color_table[code];
}

unsigned int lzw_get_bit_size(unsigned int size,
   unsigned int index)
{
   switch (index)
   {
      case 8:
      case 16:
      case 32:
      case 64:
      case 128:
      case 256:
      case 512:
      case 1024:
      case 2048:
         size++;
         break;
      default:
         return size; // No change of size.
         break;
   }

   return size;
}

/* Identifies and decodes the extension. */
void gif_decode_extension(FILE *file, struct gif_t *gif)
{
   unsigned char extension_type;
   struct extension_t *extension = (struct extension_t *)
       gif_malloc(
      sizeof(struct extension_t));
   struct extension_t *current = NULL;

   fread(&extension_type, sizeof(extension_type), 1, file);
   extension->next = NULL;

   switch (extension_type) {
      case GRAPHIC_CONTROL_EXTENSION:
         printf("Graphic Control Extension ");
         extension->type = GRAPHIC_CONTROL_EXTENSION;
         extension->extension = gif_decode_gce(file);
         printf("[decoded]\n");
         break;
      case COMMENT_EXTENSION: break;
      case PLAIN_TEXT_EXTENSION:
         printf("Plain Text Extension ");
         extension->type = PLAIN_TEXT_EXTENSION;
         extension->extension = NULL;
         gif_decode_pte(file);
         printf("[decoded]\n");
         break;
      case APPLICATION_EXTENSION: break;
   }

   // Stores the extension.
   if (gif->extensions == NULL)
      gif->extensions = extension;
   else {
      current = gif->extensions;
      while (current->next != NULL)
         current = current->next;
      current->next = extension;
   }
}

// Decodes the Graphic Control Extension.
struct graphic_control_extension_t *gif_decode_gce(FILE *file)
{
   unsigned char data;
   struct graphic_control_extension_t *gce =
      (struct graphic_control_extension_t *)gif_malloc(
         sizeof(struct graphic_control_extension_t));

   // Reads the data.
   // We don't care about block size, it's always 4 for GCE.
   fseek(file, 1, SEEK_CUR);
   fread(&data, sizeof(data), 1, file);
   gce->disposal_method = (data >> 2) & 0x07;
   gce->user_input_flag = (data >> 1) & 0x01;
   gce->transparent_color_flag = data & 0x01;
   fread(&gce->delay_time, sizeof(gce->delay_time), 1, file);
   fread(&gce->transparent_color_index,
      sizeof(gce->transparent_color_index), 1, file);
   fseek(file, 1, SEEK_CUR); // Ignore the block terminator.

   return gce;
}

// Plain Text Extension is not implemented because according to
// the GIF specification, it was officially deprecated in
// October 1998.
void gif_decode_pte(FILE *file)
```

```
{
   char identity;

   while (fread(&identity, sizeof(identity), 1, file) != 0)
      if (identity == BLOCK_TERMINATOR)
         break;
}

void *gif_malloc(size_t size)
{
   void *memory = malloc(size);

   if (memory == NULL) {
      printf("Couldn't allocate memory!\n");
      exit(1);
   }

   return memory;
}

int gif_next(FILE *file, unsigned char *character)
{
   return fread(character, 1, 1, file);
}
```

14.3 gifdecoder.h

Listing 14.3: gifdecoder.h

```
#ifndef _GIF_DECODER_
#define _GIF_DECODER_

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Macros */
#define BITGET(data, bits) (data & ((1 << bits) - 1))
#define BITGET_REMAINING(data, bits) ((data & ~((1 << bits) -
    1))\
   >> (8 - bits))

/* Flags */
#define GLOBAL_COLOR_TABLE_FLAG 0x80
#define LOCAL_COLOR_TABLE_FLAG 0x80
#define INTERLACE_FLAG 0x40
#define SORT_FLAG_G 0x08
#define SORT_FLAG_L 0x20
#define USER_INPUT_FLAG 1
#define TRANSPARENT_COLOR_FLAG 1

/* Identifiers */
#define IMAGE_SEPARATOR 0x2C
#define BLOCK_TERMINATOR 0x00
#define EXTENSION 0x21
#define GRAPHIC_CONTROL_EXTENSION 0xF9
#define COMMENT_EXTENSION 0xFE
#define PLAIN_TEXT_EXTENSION 0x01
#define APPLICATION_EXTENSION 0xFF
#define END 0x3B

/* Functions */
struct color_table_t gif_decode_ct(
   FILE *file,
   unsigned char sort_flag,
   unsigned int size
);
void gif_decode_extension(FILE *file, struct gif_t *gif);
void gif_decode_image_descriptor(FILE *file, struct gif_t *gif)
    ;
void gif_decode_lzw(
   FILE *file,
   struct gif_t *gif,
   unsigned int descriptor
);
void lzw_decode(
   struct lzw_word_table_t *table,
   unsigned char block_size,
   unsigned char *data,
   struct color_table_t *color_table
);
struct color_t get_color(
   unsigned int code,
   struct color_t *color_table
);
unsigned int lzw_get_bit_size(
   unsigned int size,
   unsigned int index
);
struct graphic_control_extension_t *gif_decode_gce(FILE *file);
void gif_decode_pte(FILE *file);
void *gif_malloc(size_t size);
int gif_next(FILE *file, unsigned char *character);

/* Structs */
struct header_t {
   char signature[3];
   char version[3];
};

struct logical_screen_descriptor_t {
   unsigned short int logical_screen_width;
   unsigned short int logical_screen_height;
   unsigned char packed_fields;
   unsigned char background_color_index;
   unsigned char pixel_aspect_ratio;
};

struct color_t {
   unsigned char r;
   unsigned char g;
   unsigned char b;
};

struct color_table_t {
   struct color_t *color_table;
   unsigned char sort_flag;
   unsigned int size;
};

struct entry_t {
   struct color_t *value;
   struct entry_t *parent;
};

struct lzw_code_t {
   unsigned short int code;
   unsigned short int bits;
};

struct lzw_word_t {
   struct lzw_code_t *code;
   unsigned short int length;
};

struct lzw_word_table_t {
   unsigned char code_size; // Size in bits of first code in
       stream.
   struct lzw_word_t words[8192]; // All stored codes.
   unsigned int number_of_words;  // Total number of codes.
   unsigned short int clear_word; // Resets the table.
   unsigned short int end_word;   // The code that ends
       decoding.
   unsigned short int start_word; // Code for the first word.
};

struct lzw_help_t {
   unsigned char buffer;
   unsigned char remaining;
   unsigned short int final;
   unsigned char more;
};

struct lzw_help_decompress_t {
   unsigned short int character;
   struct lzw_word_t word;
   struct image_descriptor_t *descriptor;
   struct color_table_t *color_table;
   unsigned short int line_length;
   unsigned int output;
   int parent;
};

struct remove_t {
   unsigned int length;
   unsigned char remove;
};

struct print_queue_t {
   unsigned int queue[8192];
   unsigned int length;
};

struct orientation_t {
   unsigned short int left;
   unsigned short int top;
   unsigned short int width;
   unsigned short int height;
};

struct image_descriptor_t {
   struct orientation_t orientation;
   struct color_table_t local_color_table;
   unsigned char interlace_flag;
};
```

```
struct graphic_control_extension_t {
   unsigned char disposal_method;
   unsigned char user_input_flag;
   unsigned char transparent_color_flag;
   unsigned short int delay_time;
   unsigned char transparent_color_index;
};

struct extension_t {
   unsigned char type;
   void *extension;
   struct extension_t *next;
};

struct descriptor_t {
   void *descriptor;
   struct descriptor_t *next;
};

struct gif_t {
   struct header_t header;
   struct logical_screen_descriptor_t logical_screen_descriptor
       ;
   struct color_table_t global_color_table;
   struct extension_t *extensions;
   struct descriptor_t *image_descriptors;
};

#endif
```

Chapter 15

Appendix E
Web Browser Image Decoding Performance Test

This is a small web page that uses some JavaScript to test the performance of the built-in image decoder in a web browser, typically written in C or C++. In this case, it's used to test the performance of GIF decoding.

N.B. You have to empty the web browsers cache before running any test!

15.1 C++ Image Decoding Speed Test using JavaScript

Listing 15.1: C++ Image Decoding Speed Test

```
<html>
<head>
   <title>C++ GIF Decoding Speed Test using JavaScript</title>

   <script type="text/javascript">
   function imageLoaded()
   {
      var stop_time = new Date;
      document.getElementById("imageContainer").appendChild(
         objImage);
      document.getElementById("log").appendChild(
         document.createTextNode("Time: " + (stop_time -
            start_time)));
   }

   function runThis()
   {
      start_time = new Date();
      objImage = new Image();
```

```
        objImage.onload=imageLoaded;
        objImage.src='images/some_image.gif';
    }
    </script>
</head>
<body onload="runThis()">

<div id="log"></div>
<div id="imageContainer"></div>

</body>
</html>
```

Chapter 16

Appendix F Patterns

16.1 General

16.1.1 NULL assignment

C++

```
global_palette = NULL;
```

JavaScript

```
global_palette = null;
```

16.1.2 Boolean values

C++

```
TRUE
FALSE
```

JavaScript

```
true
false
```

16.1.3 Changing one character

C++

```
frame_buf[frame_pos] = codes[code].data;
```

JavaScript

```
frame_buf = frame_buf.substr(0, frame_pos) +
            codes[code].data +
            frame_buf.substr(frame_pos + 1, frame_buf.length -
                frame_pos);
```

16.1.4 For-loop

C++

```
for (int line = 0; line < height; line++)
{
   ...
}
```

JavaScript

```
for (var line = 0; line < height; line++)
{
   ...
}
```

16.1.5 Assertion

C++

```
case GIF_STATE_LOCAL_COLORS:
{
   int bytes_to_copy = (1 << (local_palette_cols + 1)) * 3;
   if (numBytes - curr_pos < bytes_to_copy)
   {
      OP_ASSERT(numBytes - curr_pos >= 0);
```

JavaScript

```
// Using these functions:
function AssertException(message)
{
   this.message = message;
}

AssertException.prototype.toString = function ()
{
   return 'AssertException: ' + this.message;
};

function assert(exp, message) {
   if (!exp)
      throw new AssertException(message);
}

// Rewrite the C code to:
case GIF_STATE_LOCAL_COLORS:
{
var bytes_to_copy = (1 << (local_palette_cols + 1)) * 3; // int
   if (numBytes - curr_pos < bytes_to_copy)
   {
      assert(numBytes - curr_pos >= 0,
         "GIF_STATE_LOCAL_COLORS: Larger than or equal to 0!");
```

16.1.6 RETURN_IF_ERROR

C++

```
RETURN_IF_ERROR(SignalMainFrame(&continue_decoding));
```

JavaScript

```
// Return if error
var status = SignalMainFrame(continue_decoding);
if (status < 0)
   return status;
```

16.1.7 OP_NEWA

C++

```
local_palette = OP_NEWA(UINT8, bytes_to_copy);
if (local_palette == NULL)
{
   return OP_STATUS_ERR_NO_MEMORY;
}
```

JavaScript

```
// Just remove the code.
```

16.1.8 op_memcpy (copying of memory)

C++

```
// Example 1
op_memcpy(local_palette, (const char*)&data[curr_pos],
    bytes_to_copy);

// Example 2
op_memcpy(frame_buf + frame_pos, frame_buf + codes[code].
    cache_pos,
   codes[code].length);
```

JavaScript

```
// Example
function memcpy(dest, src, data)
{
   return data.substr(0, dest) + src + data.slice(dest+src.
       length,
           data.length);
}
```

```
frame_buf = memcpy(frame_pos, frame_buf.substr(codes[code].
    cache_pos,
   codes[code].length), frame_buf);
```

16.1.9 Opera Status Code

C++

```
#define OP_STATUS_ERR -1
#define OP_STATUS_ERR_NO_MEMORY -2
#define OP_STATUS_ERR_NULL_POINTER -3
#define OP_STATUS_ERR_OUT_OF_RANGE -4
#define OP_STATUS_ERR_NO_ACCESS -5
#define OP_STATUS_ERR_NOT_DELAYED -6
#define OP_STATUS_ERR_FILE_NOT_FOUND -7
#define OP_STATUS_ERR_NO_DISK -8
#define OP_STATUS_ERR_NOT_SUPPORTED -9
#define OP_STATUS_ERR_PARSING_FAILED -10
#define OP_STATUS_ERR_NO_SUCH_RESOURCE -11
#define OP_STATUS_ERR_BAD_FILE_NUMBER -12
#define OP_STATUS_ERR_YIELD -13

OpStatus::
   OK = 0
   ERR                    = OP_STATUS_ERR
   ERR_NO_MEMORY          = OP_STATUS_ERR_NO_MEMORY
   ERR_NULL_POINTER       = OP_STATUS_ERR_NULL_POINTER
   ERR_OUT_OF_RANGE       = OP_STATUS_ERR_OUT_OF_RANGE
   ERR_NO_ACCESS          = OP_STATUS_ERR_NO_ACCESS
   ERR_NOT_DELAYED        = OP_STATUS_ERR_NOT_DELAYED
   ERR_FILE_NOT_FOUND     = OP_STATUS_ERR_FILE_NOT_FOUND
   ERR_NO_DISK            = OP_STATUS_ERR_NO_DISK
   ERR_NOT_SUPPORTED      = OP_STATUS_ERR_NOT_SUPPORTED
   ERR_PARSING_FAILED     = OP_STATUS_ERR_PARSING_FAILED
   ERR_NO_SUCH_RESOURCE   = OP_STATUS_ERR_NO_SUCH_RESOURCE
   ERR_BAD_FILE_NUMBER    = OP_STATUS_ERR_BAD_FILE_NUMBER
   ERR_YIELD              = OP_STATUS_ERR_YIELD
   ERR_SOFT_NO_MEMORY     = -4097
```

JavaScript

```
OP_STATUS_OK = 0;
OP_STATUS_ERR = -1;
OP_STATUS_ERR_NO_MEMORY = -2;
OP_STATUS_ERR_NULL_POINTER = -3;
OP_STATUS_ERR_OUT_OF_RANGE = -4;
OP_STATUS_ERR_NO_ACCESS = -5;
OP_STATUS_ERR_NOT_DELAYED = -6;
OP_STATUS_ERR_FILE_NOT_FOUND = -7;
OP_STATUS_ERR_NO_DISK = -8;
OP_STATUS_ERR_NOT_SUPPORTED = -9;
OP_STATUS_ERR_PARSING_FAILED = -10;
OP_STATUS_ERR_NO_SUCH_RESOURCE = -11;
OP_STATUS_ERR_BAD_FILE_NUMBER = -12;
OP_STATUS_ERR_YIELD = -13;
```

16.2 Classes

16.2.1 Interfaces

C++

```
virtual void OnNewFrame(const ImageFrameData& image_frame_data)
     = 0;
```

JavaScript

```
// Types: const ImageFrameData&
// Returns:
this.OnNewFrame = function(image_frame_data)
{
   assert(false, "ImageDecoderListener: OnNewFrame: Interface!"
       );
};
```

16.2.2 Inheritance

C++

```
class ImageDecoderGif : public ImageDecoder, public LzwListener
{
};
```

JavaScript

```
// Use the following functions to implement multiple
    inheritance:
Function.prototype.DeriveFrom = function(fnSuper)
{
   var prop;
   if (this == fnSuper)
   {
      alert("Error - cannot derive from self");
      return;
   }

   for (prop in fnSuper.prototype)
   {
      if (typeof(fnSuper.prototype[prop]) == "function" &&
          !this.prototype[prop])
      {
         this.prototype[prop] = fnSuper.prototype[prop];
      }
   }
   this.prototype[fnSuper.StName()] = fnSuper;
}

Function.prototype.StName = function()
{
   var st;
   st = this.toString();
   st = st.substring(st.indexOf(" ")+1, st.indexOf("("));

   if (st.charAt(0) == "(")
      st = "function ...";

   return st;
}

Function.prototype.Override = function(fnSuper, stMethod)
{
   this.prototype[fnSuper.StName() + "_" + stMethod] =
      fnSuper.prototype[stMethod];
}

// Example
function A()
{
    this.x = 1;
}

A.prototype.DoIt = function()
{
    this.x += 1;
    alert("x: " + x);
}

function A2()
{
   this.z = 3;
}

A2.prototype.DoIt2 = function()
```

```
{
   this.z += 1;
   alert("z: " + z);
}

B.DeriveFrom(A);  // Inherits code from A
B.DeriveFrom(A2); // Inherits code from A2
function B()
{
    this.A();  // Super-class constructor (A)
    this.A2(); // Super-class constructor (A2)
    this.y = 2;
}

b = new B;
b.DoIt();
b.DoIt2();
```

16.2.3 Constructor

C++

```
LzwDecoder::LzwDecoder() : code_decoder(NULL),
   nr_of_rest_bits(0),
   rest_bits(0)
{
   ...
}
```

JavaScript

```
// Initialize private and public variables immediately when
// they're created.

...
var code_decoder = null;
this.nr_of_rest_bits = 0;
this.rest_bits = 0;
```

16.2.4 Create (definition)

C++

```
public:
static ImageDecoderGif* Create(ImageDecoderListener* listener,
   BOOL decode_lzw = true) { ... }
```

JavaScript

```
// Merge the code in Create with the Constructor.

decode_lzw = true;
```

16.2.5 Create (usage)

C++

```
if (lzw_decoder == NULL)
{
   lzw_decoder = LzwDecoder::Create(this);
   if (lzw_decoder == NULL)
   {
      return OP_STATUS_ERR_NO_MEMORY;
   }
}
```

JavaScript

```
// Just remove the code, but make sure the constructor
// initializes lzw_decoder to LzwDecoder()!
```

16.2.6 Public variable

C++

```
LzwDecoder* lzw_decoder = 0;
```

JavaScript

```
this.lzw_decoder = 0; // LzwDecoder*
```

16.2.7 Public function

C++

```
public:
static ImageDecoderGif* Test(ImageDecoderListener* listener,
   BOOL decode_lzw = true) { ... }
```

JavaScript

```
// Types: ImageDecoderListener*, BOOL
// Returns: static ImageDecoderGif*
this.Test = function(listener, decode_lzw)
{
   if (decode_lzw == null)
      decode_lzw = true;
};
```

16.2.8 Public struct

C++

```
public:
struct code {
   unsigned cache_pos = 0,
   short length = 0,
   short parent = 0,
   UINT8 data = 0,
   UINT8 first_character = 0
} codes[LZW_TABLE_SIZE];
```

JavaScript

```
// struct code { ... } codes[LZW_TABLE_SIZE]
this.code = function() {
   this.cache_pos = 0;        // unsigned
   this.length = 0;           // short
   this.parent = 0;           // short
   this.data = 0;             // UINT8
   this.first_character = 0;  // UINT8
};
this.codes = this.code[LZW_TABLE_SIZE];
```

16.2.9 Private variable

C++

```
LzwDecoder* lzw_decoder = 0;
```

JavaScript

```
var lzw_decoder = 0; // LzwDecoder*
```

16.2.10 Private function

C++

```
private:
static ImageDecoderGif* Test(ImageDecoderListener* listener,
   BOOL decode_lzw = true) { ... }
```

JavaScript

```
// Types: ImageDecoderListener*, BOOL
// Returns: static ImageDecoderGif*
var Test = function(listener, decode_lzw)
{
   if (decode_lzw == null)
      decode_lzw = true
};
```

16.2.11 Private struct

C++

```
private:
struct code {
   unsigned cache_pos = 0,
   short length = 0,
   short parent = 0,
   UINT8 data = 0,
   UINT8 first_character = 0
} codes[LZW_TABLE_SIZE];
```

JavaScript

```
// struct code { ... } codes[LZW_TABLE_SIZE]
var code = function() {
   this.cache_pos = 0;       // unsigned
   this.length = 0;          // short
   this.parent = 0;          // short
   this.data = 0;            // UINT8
   this.first_character = 0; // UINT8
};
var codes = code[LZW_TABLE_SIZE];
```

Chapter 17

Appendix G

17.1 Test and Development Environment

17.1.1 Laptop

CPU: 2.5 GHz Intel Core 2 Duo T9400
Memory: 4 GB DDR2 RAM
Hard drive: 200 GB 7200 rpm
Operating systems: Windows 7 Ultimate, Ubuntu Linux 9.10

17.1.2 Tools

- Eclipse build 20090920-1017
- GCC 4.4.1
- GIT 1.6.3.3
- KDevelop 3.9.95
- Visual Studio 2008 Express Edition
- Valgrind 3.5.0

17.1.3 Web Browsers

These are the web browsers that my code were tested against. However, if not explicitly written otherwise, only the latest versions of IE, Chrome, Firefox and Opera were tested.

- Firefox 2.0.0.20 (2009-01-08)
- Firefox 3.0.17 (2010-01-05)
- Firefox 3.6.2 (2010-03-22)
- Google Chrome 4.1.249.1042 (2010-03-24)
- Internet Explorer 5.5 (2000-07-??)
- Internet Explorer 6.0 SP3 (2008-05-05)
- Internet Explorer 7.0 (2008-12-17)
- Internet Explorer 8.0 (2009-03-19)
- Opera 6 (2003-03-24)
- Opera 7.54 (2004-08-04)
- Opera 8.54 (2007-03-28)
- Opera 9.64 (2009-03-03)
- Opera 10.51, (2010-03-22)
- Safari 3.2.3 (2009-05-12)
- Safari 4.0.5 (2010-03-11)

Chapter 18

Appendix H

18.1 SunSpider

The descriptions of the test suite are cited from from dromeo.com.

3D Mesh Transformation

Transforming the points of a matrix. Tests: array, looping, math.

3D Ray-trace

Rendering a scene using ray-tracing techniques. Tests: array, functions, math.

AES Encryption/Decryption

Encrypt a string and then decrypt it again using AES. Tests: looping, string, bit-ops.

Bit-wise And

Compute a number by using a series of 'and' bit operations. Tests: looping, bit-ops.

Compute Bits in Byte

Compute the number of bits in a number using bit-ops. Tests: looping, bit-ops.

Compute Bits in Byte (2)

Compute the number of bits in a number using bit-ops. Tests: looping, bit-ops.

DNA Sequence Alignment

Find DNA matches within a larger sequence. Tests: string, object, looping.

DNA Sequence Counting

Counts occurrences in a DNA sequence. Tests: string, regexp.

Date Formatting

Converting a date into a string representation. Tests: date, string.

Date Formatting (2)

Converting a date into a string representation. Tests: date, string.

Fannkuch

Figure out the number of ways in which a set of numbers can be manipulated. Tests: array, looping.

MD5 Hashing

Hash a long string using MD5. Tests: looping, string.

N-Body Rotation and Gravity

Compute the location of multiple planets based upon rotation and gravity. Tests: object, property, looping, math.

Partial Sum Calculation

Calculate the partial sum of a few different number series. Tests: math, looping.

Prime Number Computation

Compute the number of prime numbers in a specific range of numbers. Tests: looping, array.

Prime Number Computation (2)

Compute the number of prime numbers in a specific range of numbers using bit operations. Tests: looping, bit-ops, array.

Recursive Number Calculation

Compute various numbers in a recursive manner. Tests: functions.

SHA1 Hashing

Hash a long string using SHA1. Tests: looping, string.

Script Unpacking

Decompressing scripts run through Dean Edwards' Packer. Tests: regexp, string, looping.

Spectral Norm of a Matrix

Calculate the spectral norm of a matrix of numbers. Tests: array, looping, math.

Tag Cloud Creation

Convert a JSON structure into an HTML tag cloud. Tests: string, regexp.

Traversing Binary Trees

Moving through an object representation of a binary tree. Tests: object, recursion.

Trigonometric Calculation

Calculate values from hyperbolic and trigonometric functions. Tests: math, looping.

Validate User Input

Test user input against a series of rules. Tests: string, regexp.

18.2 V8 Benchmark Suite

The descriptions of the test suite are cited from from
http://v8.googlecode.com/svn/data/benchmarks/v5/run.html.

Richards

OS kernel simulation benchmark.

DeltaBlue

One-way constraint solver.

Crypto

Encryption and decryption benchmark, using RSA.

RayTrace

Ray tracer benchmark.

EarleyBoyer

Classic Scheme benchmarks.

RegExp

Regular expression benchmark generated by extracting regular expression operations from 50 of the most popular web pages.

Splay

Data manipulation benchmark that deals with splay trees and exercises the automatic memory management subsystem

Chapter 19

Appendix I

19.1 Creating the Method used while Implementing a GIF Decoder

The method-flow used for implementing the tasks were developed after lots of trial and error, trying out different approaches.

First, I tried an evolutionary approach using the C programming language. I began implementing the GIF decoder after having read one third of the specification, the amount I thought was enough. Soon problems arose. The main issue with using the evolutionary approach was extending the code. When it turned out that more of the specification had to be implemented, the code was so tightly coupled that the new parts didn't fit in. The architectural design had to be rethought, the code had to be redesigned and it ended up being easier to just rewrite the decoder from scratch, rethinking the whole implementation.

Second, it was a mistake to use a complex programming language I didn't fully understand. Instead of being able to focus on solving the specific problem, a lot of time went to debug memory allocation, correcting functions with unexpected behavior, fixing pointers and making sure the read input was correct.

Third, from the very beginning I was focusing on speed, memory usage and design, all at once, instead of only implementing a working decoder.

Forth, before testing that the reading of the GIF data was done right, I began implementing the LZW decompression. The same error was remade. I didn't fully understand the algorithm, but stubbornly implemented a decoder for what I thought was LZW. We'll never know if my understanding was correct, because the combined errors from the GIF and LZW decoding was too many.

In conclusion, a systematic approach must always be used, doing one step at a time. Thorough research is extremely important, by reading the specification carefully and gather additional knowledge from other sources.

Using a less complex language like Python or Pike for the first implementation helps, as one will quickly discover errors in the comprehension of the problem. The first implementation should systematically focus on only implementing a working decoder. Speed, memory usage and number of code lines are of secondary concern. Lastly, creating and using lots of test cases are of great importance. To test module behavior, their coupled behavior and rerun the tests after optimizing or changing a module identifies most errors right after they were introduced.

When I implemented the first version of the GIF decoder I made lots of mistakes, and that was deeply beneficial. It turned out to be a great way of learning C++, learn about memory management, pointers, repeat my understanding of data structures and how to debug memory leaks with tools like Visual Studio, Valgrind and GDB. Also, how to use an HEX editor to follow the GIF stream, understanding the GIF specification and to find out why my code sometimes only decoded part of the image. Making all these mistakes made me learn why a strict systematic approach is vital in problem solving. One can read books and papers about the importance of using a systematic method when solving a problem, but the real understanding why it's important comes from doing at least some of those mistakes by yourself, getting the core knowledge of how much time often is wasted by bad planning and implementation.

Glossary

AJAX
: Asynchronous JavaScript and XML. The technology that makes it possible to develop interactive communicating web applications.

Chrome
: Web browser created by Google.

CSS
: Cascading Style Sheets is a style sheet language used to describe the look and formatting of a document written in a markup language.

DOM
: Document Object Model. A way to refer to XML or XHTML/HTML elements as objects.

Firefox
: Web browser created by Mozilla Foundation.

Flash
: Flash, or Adobe Flash, is a multimedia platform commonly used on web pages to enhance interactiveness, display animated advertises, video and audio.

GDB
: GNU Debugger. A debugger for Windows and Unix-like systems.

GIF
: Graphics Interchange Format. A 256 color graphics format.

HTML
: Hyper Text Markup Language. The dominating language to describe the layout and content of a web page.

IDE
: Integrated Development Environment. Is normally a collection of a source code editor, compiler or parser, tools to automate building the source and a debugger.

IE
: Internet Explorer. Web browser created by Microsoft.

Java
: Object-oriented programming language invented by Sun Microsystems.

JavaScript
: In this thesis, JavaScript or JS means ECMA Script, standardized by Ecma International in the standard ECMA-262.

JIT
: Just in time compilation means a process where the engine running the code both produces bytecode but also native machine code. This is done by using various heuristics to identify code and functions that would benefit greatly from being compiled into native machine code.

JS
: See JavaScript.

JScript
: Microsoft's dialect of JavaScript.

JSON
: JavaScript Object Notation, an open lightweight standard to represent data structures or associative arrays in a human readable format. Primarily used to transmit data between web server and AJAX web applications.

Lint checking
: In this thesis, lint checking means a process where a software parses through the source code looking for syntax errors and other potentially wrong code lines.

LZW
: Lempel-Ziv-Weich. Universal loss-less data compression algorithm.

Opera
: Web browser created by Opera Software.

Profiler A profiler is a performance analysis tool that measures the frequency and duration of function calls.

Safari
: Web browser created by Apple.

Silverlight
: Microsoft's answer to Flash, with similar features and use cases.

Valgrind
: A tool used for debugging code, profiling and to aid in identifying memory leaks on Unix-like systems.

Visual Studio
: Microsoft's popular Windows IDE for C++, C# and others.

VM
: Virtual Machine

W3C
: World Wide Web Consortium is the main international standards organization for the World Wide Web.

WebGL
: Web Graphics Library is a not finished standard specification defining a JavaScript API for writing web applications utilizing hardware accelerated 3D graphics.

WHATWG
: Web Hypertext Application Technology Working Group. A group of people with interest to develop HTML and related technologies. Supported by individuals from companies such as Apple, Mozilla Foundation, Opera Software and Google.

XHTML
: Extensible Hypertext Markup Language, an extension to HTML.

XML
: Extensible Markup Language. A collection of rules that describes how to encode electronic documents. This thesis use its own interpretation, close to the HTML 4.01 standard.

Bibliography

[1] B. Wellman and K. Hampton, “Living networked on and offline,” *Contemporary Sociology*, vol. 28, pp. 648–654, November 1999.
http://www.jstor.org/stable/2655535
ISSN: 00943061.

[2] D. Hachamovitch, “Internet explorer 8 and acid2: A milestone.” Microsoft’s IEBlog, December 2007.
http://blogs.msdn.com/ie/archive/2007/12/19/internet-explorer-8-and-acid2-a-milestone.aspx.

[3] J. J. Garrett, “Ajax: A new approach to web applications.” Adaptive Path Ideas: Essay Archives, February 2005.
http://www.adaptivepath.com/ideas/essays/archives/000385.php.

[4] A. Swartz, “A brief history of ajax.” Raw Thought, December 2005.
http://www.aaronsw.com/weblog/ajaxhistory.

[5] A. Ritter, “A jpeg encoder for javascript.” source code, November 2009.
http://www.bytestrom.eu/blog/2009/1120a_jpeg_encoder_for_javascript.

[6] B. Firshman, “Jsnes - a javascript nes emulator.” source code, January 2010.
http://benfirshman.com/projects/jsnes.

[7] D. E. Liddle, “The impact of moore’s law,” *Solid State Circuits*, September 2006.
http://www.ieee.org/portal/pages/sscs/06Sept/Liddle.html.

[8] P. Thiemann, *Towards a Type System for Analyzing JavaScript Programs*, vol. 3444/2005. Springer Berlin / Heidelberg, March 2005. DOI: 10.1007/b107380, ISSN: 0302-9743, ISBN: 3-540-25435-8.

[9] T. O’Reilly, “What is web 2.0: Design patterns and business models for the next generation of software,” *International Journal of Digital Economics*, vol. 65, pp. 17–37, March 2007.

[10] P. Krill, "Javascript creator ponders past, future," *InfoWorld Developer_World*, June 2008.
http://www.infoworld.com/d/developer-world/javascript-creator-ponders-past-future-704.

[11] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Characterizing real-world behavior of javascript programs," *Microsoft Research Technical Report*, vol. MSR-TR-2009-173, December 2009.

[12] I. Summerville, *Software Engineering: (Update)*. Pearson Education Limited, eight ed., June 2006.
ISBN: 9780321313799.

[13] O. S. A. Apple Computer Inc., Mozilla Foundation, "Web workers - draft recommendation," January 2010.
http://www.whatwg.org/specs/web-workers/current-work.

[14] V. Ozcelik, "Memory leakage in internet explorer - revisited," December 2007.
http://www.codeproject.com/KB/scripting/leakpatterns.aspx.

[15] T. Mikkonen and A. Taivalsaari, "Using javascript as a real programming language," *SML Technical Report Series*, vol. 168, October 2007.
http://research.sun.com/techrep/2007/smli_tr-2007-168.pdf.

[16] P. Ratanaworabhan, B. Livshits, D. Simmons, and B. Zorn, "Jsmeter: Measuring javascript behavior in the wild," *Microsoft Research Technical Report*, vol. MSR-TR-2010-8, January 2010.

[17] A. Eid and F. Al-Dhelaan, "Google chrome browser: a security and performance study," December 2008.

[18] J. Nielson, C. Williamson, and M. Arlitt, "Benchmarking modern web browsers," *Proceedings of IEEE HotWeb 2008*, October 2008.

[19] M. Richards, J. Maloney, M. Wolczko, T. Wu, A. Burmister, and Google, "V8 benchmark suite - version 5." source code, July 2009.
http://v8.googlecode.com/svn/data/benchmarks/v5/run.html.

[20] Apple, "Sunspider javascript benchmark 0.9." source code, December 2007.
http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[21] C. Bueno, "Measuring javascript parse and load." source code, February 2010.
http://carlos.bueno.org/2010/02/measuring-javascript-parse-and-load.html.

[22] D. B. et al., “The great win32 computer language shootout.” source code, April–April 2002–2008.
http://dada.perl.it/shootout/.

[23] StatCounter, “Statcounter global stats.” web page, March 2010.
http://gs.statcounter.com.

[24] T. Software, “Tiobe programming community index for january 2010,” January 2010.
http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[25] D. Jang and K.-M. Choe, *Points-to analysis for JavaScript.* New York, NY, USA: ACM, 2009.
DOI: 10.1145/1529282.1529711, ISBN: 978-1-60558-166-8.

[26] P. M. Lionbridge, VeriTest, “i-bench 5.0.” Image, November 2003.
ftp://ftp.pcmag.com/Benchmarks/i-bench/.