Performance Analysis and Acceleration for Rich Internet Application Technologies*

Timo Ernst†

Prof. Dr. Franz Schweiggert
Dr. habil. Johannes Mayer
Dr. Norbert Heidenbluth

University of Ulm‡

May 9, 2011

*Version 1.11
†http://www.timo-ernst.net
‡Institute of Applied Information Processing

I hereby declare that the whole of this diploma thesis is my own work, except where explicitly stated otherwise in the text or in the bibliography.

Ulm, July 15th. 2010

Timo Ernst

I'd like to thank everyone from the Institute of Applied Information Processing at the University of Ulm, especially Prof. Dr. Franz Schweiggert as well as Dr. habil. Johannes Mayer for their expertise, Dr. Norbert Heidenbluth for mentoring my work and Dipl.-WiWi Steffen Fritzsche for the inspiring discussions.

Furthermore, I thank Dipl. M.S. Designer Andreas Ritter for the help on the JavaScript-version of the JPEG-encoder, August Lammersdorf from InteractiveMesh.com for the informations on the technical details of his FXCanvas3D component for JavaFX as well as Dipl. Inf. Oliver Gableske and Dipl. Inf. Alexander Forschner for their advice in general.

Abstract

This thesis introduces a set of performance tests for Rich Internet Application technologies like Adobe Flex, JavaFX, Silverlight and JavaScript. Testing categories include data-stream manipulation, cryptographic en-/decoding, mathematical operations and other relevant comparisons. The goal is to test how each of the currently available RIA technologies perform in various situations. The whole test is basically split into two types of benchmarks:

1. Use-case tests
    - API-test
    - Non-API test
2. Focus-Tests

As the name already says, use-case tests are based on a use-case like e.g. prime number generation or JPEG-compression. Since it is possible that performance-losses can be caused through the RIA runtime itself or the API used, this test is divided into API- and Non-API-tests in order to examine where slowdowns occur. API-tests are based on as many API-calls as possible, while Non-API tests mostly rely on self-written algorithms.

As a last step, all the values gathered from the use-case benchmarks lead to the creation of the so called »Focus-tests« which can be used to examine suspicious results in a more detailed way.

Example: In this thesis, a use-case test, which generates 20000 prime numbers and stores them all one by one into an array, will be introduced. The version for JavaFX showed a heavy slow-down while other benchmarks related to this cryptographic context did perform well. Thus, it was assumed that the insert-operation for arrays could be the cause for this performance loss. In order to further investigate, a dedicated test for arrays was included to the series of Focus-tests and it could be proved that the priorly made assumption, about the inefficiency of insert-operations for JavaFX sequences, was correct.

Furthermore, based on the results of these tests, a new technique called *Jaction* for boosting Flash-based application performance by using JavaScript technology will be introduced together with a dedicated framework for this technique. The basic idea is to partially delegate application logic from Flash to the »outer« JavaScript engine of the surrounding browser in order to benefit from its fast engine. In a demo which compresses a PNG image using

Jaction and the JPEG encoding algorithm, it could be shown that a significant acceleration is possible especially on Webkit-based browsers, like Google Chrome or Apple's Safari since their JavaScript engines are fast enough. Other browsers like Firefox or Internet Explorer are currently to slow to benefit from Jaction. Opera is not compatible at all due to problems regarding the JavaScript-calls from within Flash. Therefore it is very import to use the framework provided through this thesis, which takes care about these cross-browser issues.

Contents

1 Introduction

1.1 The evolution of the World Wide Web

The history of the WWW starts back in the 1960's with the development of the *ARPANET*[1] by the United States Department of Defense. The idea was to create a network which allowed universities, which were researching for the U.S. military, to transfer data across large distances. Later, this network was opened to the rest of earth's population. The Internet was born.

In the year of 1980, Tim Berners Lee invented the so called HTTP[2] protocol as well as the markup language HTML[3], which are still the basis for today's WWW. Unfortunately, both terms »Internet« as well as »World Wide Web« are often mistakenly used as synonyms, which is not correct. The Internet is a global computer network system which connects multiple hosts together so they are able to share data among each other. The WWW is just an operation on the Internet defined through the HTTP protocol, like for example FTP[4] or e-mail, based on the idea of *request* and *response*. A client can communicate with a server by sending a message (=the request) to it which can then reply (=the response) with the desired content. In earlier days of the World Wide Web, this method lead to many so called site-refreshes. For each response the server sends to the client, the full web-site must be re-loaded. This lead to high loading times because often a lot of data, which was already received in a prior response, had to be re-sent again. This redundancy is inefficient and often leads to low usability since users must wait longer than necessary for a web-site to refresh. This changed with the idea of Ajax[5], developed by Jesse James Garrett, a user experience designer from Florida, back in 2005. This technology allows web-developers to asynchronously request data from services and then manipulate the DOM[6] tree (which represents the structure of web-sites) in order to only modify the part of the web-site which should display the new content and thus minimize the number of required screen-refreshes. This approach is usually based on the JavaScript programming language, which is supported in all commonly

[1] Advanced Research Projects Agency Network
[2] Hypertext Transfer Protocol
[3] Hypertext Markup Language
[4] File Transfer Protocol
[5] Asynchronous JavaScript And XML
[6] Document Object Model

used web-browsers. Today, Ajax is very popular since it can increase the usability of web applications and lower the amount of data which has to be transferred over the internet, at the same time. [Web09]

Although the term *Ajax* sounds like a specific technology, it actually is just a concept, which can not only be used with JavaScript. In the last years, proprietary technologies like Adobe's Flash platform or Microsoft's Silverlight were attempts to transfer this idea of Ajax to plugin-based techniques, which claimed to not have the drawbacks JavaScript has, like for example various cross-browser issues caused through different implementations of web-standards by the browser manufacturers. These technologies also often provide an API with additional functionality in order to create web-applications, which can do more than just display data. This new generation of so called »webapps« often implement more application logic to the client system than prior versions, which leads to two important benefits. One is that the number of requests to the server gets minimized because no extra call is necessary if the client can take care about it locally. This can reduce load on the back-end systems. The other is that the response-time of the user-interface gets minimized since fewer server-requests are required, which again increases usability.

Example: A web-application representing a simple calculator without Ajax would have to poll the server for each mathematical operation (like e.g: 3+5 =?) and then reload the entire website with the markup received from the server reponse. With Ajax, the client sends a HTTP request to the back-end and achieves only the result *8* (usually in XML or JSON form) and nothing more. Then, the DOM-tree of the page would be modified by displaying this value through the usage of JavaScript. Thus, the only purpose of the server for the call is the calculation of 3 + 5. No additional markup would have to be re-sent (no matter if it would be redundant or not) and the client can offer an increased user-experience since only a part of the website would have to be re-loaded. It is also clear that server load would dramatically get reduced and response time improved whenever the user wants to make a calculation.

This better possibility to interact with web-applications is often referred as »richness«, which lead to the term of *Rich Internet Applications*.

1.2 Rich Internet Applications: Definition

> »The phrase *Rich Internet Application* is a lot like the word *pornography*. Easy to identify but hard to define.«
>
> *(William Grosso, java.net)* [Gro05]

As William Grosso mentioned in his humorous statement about *Rich Internet Applications* (also simply called »RIA's«), defining this term is not easy. If one searches for a bulletproof definition of this word, the chance of being disappointed is pretty high. The following lists some attempts:

> »Rich Internet applications (RIAs) attract site owners' attention because users like them, they enable interactions that HTML can't, and they get results. (...)« *(Ron Rogowski)* [Rog07]

> »Macromedia defines RIAs as combining the best user interface functionality of desktop software applications with the broad reach and low-cost deployment of Web applications and the best of interactive, multimedia communication. (...)« *(Joshua Duhl)* [Duh03]

> »Rich Internet applications (RIAs) offer a rich, engaging experience that improves user satisfaction and increases productivity. Using the broad reach of the Internet, RIAs can be deployed across browsers and desktops« *(Adobe Systems Inc.)* [Ado]

> »Rich Internet Applications (RIAs) are web applications that have most of the characteristics of desktop applications, typically delivered either by way of a standards based web browser, via a browser plug-in, or independently via sandboxes or virtual machines. Examples of RIA frameworks include Ajax, Curl, GWT, Adobe Flash/Adobe Flex/AIR, Java/JavaFX, Mozilla's XUL, OpenLaszlo and Microsoft Silverlight« *(Wikipedia)* [Wik]

> »RIA's are:
>
> - Web-based applications with desktop-like functionality
> - A slippery-slope definition«
>
> *(Andrew Kirkpatrick)* [Kir09]

As denoted by Kirkpatrick, there is no real accepted definition on RIAs which often leads to discussions, like for example: »Is iTunes a Rich Internet Application?«. On one hand, it definitely uses the Internet as a data transportation platform and offers the user a »rich user experience«. On the other hand, it cannot be called a web application, since it does not run inside a browser or uses any kind of web (frontend-)technology, like a virtual machine or plug-in. If the result of this discussion was that iTunes cannot be called a RIA: What if it would be re-written using Adobe's AIR platform, which claims to be a special runtime environment for Rich Internet Applications?

As seen in this example, defining RIAs is not easy. For the purpose of this thesis, the term *Rich Internet Application* will be, similar to the common understanding about thin, fat and rich clients, successively defined as follows:

> **Definition.** A *Thin Application* is a program, which offers only very basic functionality in order to fulfill its original purpose and nothing else.
>
> Example: Notepad
>
> **Definition.** A *Thick Application* is the opposite of a Thin Application, which means that Thick Applications offer more functionality than absolutely necessary. This is usually done in order to increase the application's usability and speed up the workflow.
>
> Example: Notepad++
>
> **Definition.** A *Rich Application* is a special subtype of Thick Applications. The difference between both is that Rich Applications offer (»rich«) functionality beyond the purpose they were made for.
>
> Example: Notepad++ with a built-in file-system management tool
>
> **Definition.** A *Rich* ***Internet*** *Application*, also called RIA, is a **Rich Application** which is related to the context of the internet regarding the way it is being deployed and/or communicating with remote hosts.
>
> Example: Notepad++ with built-in FTP-client functionality

Based on this definition, it must be said that iTunes *is* a Rich Internet Application, since it offers a lot more functionality (e.g. iPhone synchronization, podcast subscription, web-radios, iTunes store etc...) than compared to its original purpose as a pure media-player and -management application and additionally it uses the internet for media purposes like e.g. music downloads.

Note:

- Some Rich Internet Applications do not use the internet as a platform for data-transport, but are being deployed inside an internet-related technology, like a web-browser for example.

  Example: A photo-editor, which is being downloaded over the internet and deployed inside a web-browser (with or without a plug-in). Once it is deployed, it

could be used without an internet connection as long as no communication with a backend-system is necessary.

- Some internet applications are also called RIAs because they offer the user a wide range of possibilities to interact with the user-interface. This also can be stated as »richness« regarding usability and is thus conform with the above definition.

  Example: A dynamic version of a static website offering the user more interactivity using technologies like JavaScript and Ajax though visual instruments like fade-animations or hover-effects.

1.3 RIA technologies

By the time this thesis was written, four RIA technologies were competing against each other, which are namely Adobe Flex, Sun JavaFX, Microsoft Silverlight and JavaScript. There are more alternatives currently available like ZK[7] for example, but since this and other similar platforms are built on JavaScript front-end technology, their examination becomes redundant because plain JavaScript code is being tested in this thesis anyway. Thus, it was decided to only compare the already mentioned four RIA runtimes. The following sub-sections will give a short introduction to these platforms along with some historical background as well as simple examples in order to get an idea how these technologies work. Thus, the next pages will provide a fairly high amount of listings containing sample source-code.

1.3.1 Adobe Flex

Flex is an open-source RIA-framework built by Adobe Systems Inc. on top of the Flash platform. The Flash Player itself, which is required to playback any kind of Flash content, is a proprietary technology although Adobe released the specification for SWF[8], the container format for Flash applications, as an open document. Flash was originally developed by a company called Macromedia (which was later acquired by Adobe Systems Inc.) in order to bring animations (including sound) to the World Wide Web. It was never intended to be a platform for Rich Internet Applications, but through the evolution of the web, being pushed forward through a huge community of creative Flash-developers and -designers, this technology became almost ubiquitous in the WWW since it is very often used and required if it comes up to playback of multi-media content of any kind, like for example videos and animations. In order to view Flash content on websites,

[7] http://www.zkoss.org/
[8] Shock Wave Format

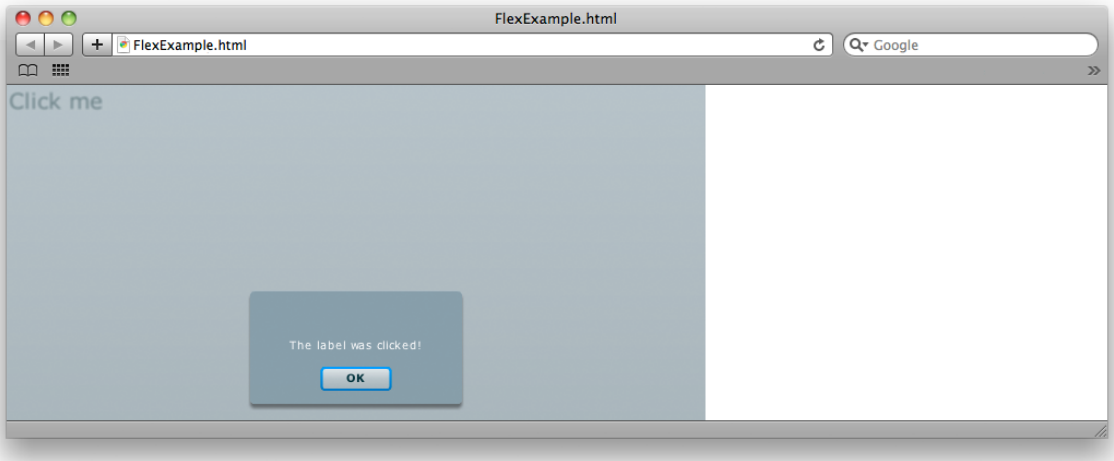

Figure 1.1: The compiled version of the Flex-example using a custom label-component

browsers require a plug-in, which needs to be installed on the user's operating system. Flash-applications which are targeted for the desktop must be compiled as an so called »AIR[9]-application«, which can be locally installed on a user's computer and run inside the Adobe AIR Runtime Environment, quite similar to Java applications, deployed on the Java Virtual Machine. Flash applications are being developed using a scripting-language called ActionScript, which is currently in version 3. Since its early moments, AS3[10] has made huge progress and can be called a fully object-oriented programming language including the whole concept of polymorphism utilizing interfaces, (abstract) classes and more. With the release of the Flex framework, Adobe introduced a new XML-application called MXML, which is a descriptive markup language for graphical user-interfaces in Flash, while the core-application logic is still being written in ActionScript3. For a better understanding of the technology, the combination of MXML and AS3 can be compared to websites created with HTML (for the GUI[11]) and JavaScript (for the application logic). Each UI[12]-component in Flex belongs to `mx.*` or a sub-package of it. The elements included have all one thing in common: Every Flex-component associated with user interfaces extend the `UIControl` class, which again extends other super-classes itself, like the `<mx:Text />`-component for example:

```
Text extends Label
extends UIComponent
extends FlexSprite
extends Sprite
extends DisplayObjectContainer
```

[9] Adobe Integrated Runtime
[10] ActionScript3
[11] Graphical User-Interface
[12] User-Interface

```
extends InteractiveObject
extends DisplayObject
extends EventDispatcher
extends Object
```

This illustrates very well how Flash is not as bad as its reputation when people blame it to be only a tool for designers. Actually, the opposite applies. Flash strictly implements the whole concept of object-oriented programming and offers a very consistent platform for developers. The following, very simple example shows how MXML and ActionScript3 work together in an object-oriented way. Please note that `import`-statements were omitted whenever possible since space is limited in this thesis.

The intention of the code snippet below is to extend the `mx.controls.Label` component, which is normally used for displaying short text data, and give it the ability to be clicked. Once that happens, an alert-box will pop up confirming the user-interaction. This custom component class should look pretty familiar to developers with a Java-background, except for minor differences regarding syntax:

```
1  package {
2   // Import-statements omitted
3
4   // A new component-class which extends mx.controls.Label
5   public class ClickableLabel extends Label {
6
7    public function ClickableLabel() {
8     super(); // Call the super constructor
9
10    // Add a click-listener to this component
11    addEventListener(MouseEvent.CLICK, labelClicked);
12   }
13
14   // Listener-function which gets called once the user clicks on
15   // this component
16   private function labelClicked(event:MouseEvent):void {
17    Alert.show("The label was clicked!");
18   }
19  }
20 }
```

Listing 1.1: File »ClickableLabel.as«

A very special feature of the Flex framework is the markup language *MXML*, which can help to save a lot of coding work. The above example of ClickableLabel.as can also been expressed as follows:

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <mx:Label click="mx.controls.Alert.show('The label was clicked!');"
3     xmlns:mx="http://www.adobe.com/2006/mxml">
4 </mx:Label>
```

Listing 1.2: File »ClickableLabel.mxml«

These four lines of XML are absolutely equivalent to the AS3-example (ClickableLabel.as) above but drastically reduce the amount of code through the usage of declarative syntax instead of imperative programming techniques. This shows how easy it can be to extend an existing component and add a click-listener to it through the XML-attribute »click«, which again defines the AS3 code, that should be executed once the event fires. The Flex framework compiler is able to transform this MXML into plain ActionScript3 code, which can be interpreted by the Flash Player runtime, which does not have the ability to understand MXML. The benefit of this technique is generated (and thus highly efficient) AS3 code and a minimum of MXML markup, which can dramatically reduce coding effort while increasing runtime performance at the same time.

The main class for this demo application, which is being instantiated on application startup, is written in MXML. Similar to HTML, the custom component `ClickableLabel` is appended to the main stage by simply adding a new XML element, which automatically instructs the Flash Runtime Environment to create an instance of the class.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
3     layout="absolute"
4     xmlns:local="*"
5     width="640"
6     height="480">
7  <local:ClickableLabel text="Click me" fontSize="20"/>
8 </mx:Application>
```

Listing 1.3: File »FlexExample.mxml«

See figure 1.1 on page 14 for a screenshot of the running application after the label was clicked by a user.

1.3.2 Oracle/Sun JavaFX

JavaFX is an open-source technology built on top of the Java Virtual Machine, developed by Chris Oliver who used to work at a company called »See Beyond Technology Corp« which got acquired by Sun/Oracle in the year of 2005[Mor10]. This platform claims to offer the whole API of the well-known Java-world to RIA-programmers. JavaFX developers create applications using JavaFX Script, a programming language created only for this purpose, which is a mix of traditional script-coding techniques and a new declarative approach for creating user-interfaces.

For example, the following code snippet, shown in listing 1.4 on page 17, creates a simple hello-world application (See figure 1.2 on page 18 for a screenshot of the compiled and running version).

```
1  // "Ordinary" definition of a method in JavaFX
2  function getTextToShow():String{
3   return "Hello World";
4  }
5
6  // Declarative description of the user-interface
7  Stage {
8      title: "My first JavaFX application"
9      scene: Scene {
10         width: 640
11         height: 480
12         content: [
13             Text {
14                 font : Font {
15                     size : 16
16                 }
17                 x: 10
18                 y: 30
19                 content: getTextToShow();
20             }
21         ]
22     }
23 }
```

Listing 1.4: Creation of a graphical user-interface in JavaFX

This integration of a declarative approach into scripting languages is new. Until now, these kinds of constructs were only known from technologies utilizing two languages, which are usually a mix of XML and an imperative programming language, like for example MXML+ActionScript3 in Adobe Flex. This attempt of Sun Microsystems finally removes the border between both approaches and integrates them into one language. Furthermore, like in many other (RIA-)technologies, JavaFX finally introduces the concept of data-binding to the Java-world. The following (simplified) example illustrates this:

```
1  Stage {
2      title: "My first JavaFX application!"
3      scene: Scene {
4          content: [
5              Text { content: bind textToDisplay }
6          ]
7      }
8  }
9  var textToDisplay:String = "Hello world";
10 textToDisplay = "modified";
```

Listing 1.5: File »Main.fx«

Figure 1.2: The compiled version of the JavaFX HelloWorld-example

The JavaFX code snipped above basically creates the same application as in the prior example, but this time, a string-variable is bound to the text component. Whenever the content of the variable `textToDisplay` changes, the bound UI component will get updated. Thus, the string »modified« will be displayed instead of »Hello world«. This technique requires less effort for developers since they won't have to bother with implementing this behaviour on their own.

Another great feature is the possibility to integrate classic Java code into JavaFX applications. Sun claims that both languages interact well together, which means that it is possible to access Java methods and classes from within JavaFX files and vice versa. The following short example will demonstrate how a float-value will be rounded using:

1. The JavaFX class `javafx.util.Math`
2. The traditional Java class `java.lang.Math`
3. An own custom class written in plain Java

The example source code looks like this:

```
1 package test;
2 // Traditional Java code
3 public class JavaTestClass {
4  public int round(float value){
5   return Math.round(value);
6  }
7 }
```

Listing 1.6: File »JavaTestClass.java«

```
1  package test;
2  var valueToRound:Float = 1.3554;
3
4  // Use JavaFX API
5  var result:Integer = javafx.util.Math.round(valueToRound);
6  println("JavaFX API: {result}");
7
8  // Use classic Java API
9  result = java.lang.Math.round(valueToRound);
10 println("Java API: {result}");
11
12 // Use custom Java class
13 var javaInstance:JavaTestClass = new JavaTestClass();
14 result = javaInstance.round(valueToRound);
15 println("Custom Java class: {result}");
```

Listing 1.7: File »Main.fx«

The result of the code above looks as follows:

```
JavaFX API: 1
Java API: 1
Custom Java class: 1
```

Note: *Since JavaFX does not overload the (+)-operator for string-concatenation, an interpolation technique with curly brackets {} is used here.*

The prior example shows very well how JavaFX and Java code can interact between each other. Beyond this, it is also possible to use classic Java GUI components, for example controls from the common Swing package, inside JavaFX applications. Although there are some limitations to this (e.g. using Java3D, see 2.3.1.3 on page 49), this usually works well with standard UI controls.

JavaFX earns most critics when it comes up to loading times (see figure 1.3 on page 20). Similar to classic Java applets, JavaFX applications tend to load very slowly until they are ready to use. Furthermore, the component set for creating graphical user-interfaces is still very small. There have been some enhancements from version 1.2 to 1.3 but this is still way less than compared to what other RIA technologies like Flex oder Silverlight have to offer. Besides these drawbacks, JavaFX is a very promising technology, even if it is still in its early days. The new declarative approach combined with Java's huge API definitely is an interesting combination.

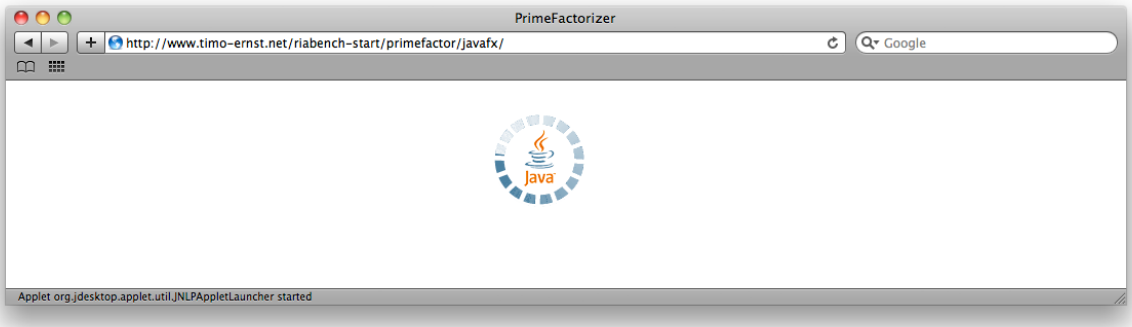

Figure 1.3: A typical JavaFX situation: Waiting for the application to load

1.3.3 Microsoft Silverlight

Silverlight is a highly proprietary RIA solution by the Microsoft Corporation as a counter-product to Flash. Similar to Adobe's platform, Silverlight also relies on a plug-in, that has to be installed on the user's operating system, which is already set up on newer versions of Microsoft Windows. This definitely frees the user from downloading the plug-in on their own which is often a barrier between Flash and its users.

Silverlight itself belongs to the .NET family, which means that both, XAML (a XML-based markup language similar to MXML) as well as the CLI[13] are supported. This enables experienced developers, who are familiar with .NET, to easily start their first Silverlight projects. Usually XAML is being used together with C# but it is also possible to use Visual Basic, J# or any other CLI-compatible programming language. Each XAML component is being stored inside a *.xaml file which is always associated with another file containing code written with an imperative programming language. Both are inseparably connected to each other in order to clearly separate view- from controller-components.

For example: The following hello-world XAML-component has a connected C# class, which interacts as the controller in order to manipulate the view as well as to catch user interaction events. The XAML view-component only defines the way, the application should look like and adds a click listener to a given button component. Once the button gets clicked, the associated controller-class (HelloWorld.cs) will catch the event. See the listings on page 21 for the example source-code and figure 1.4 on page 22 for the screenshot of the running application.

[13] Common Language Infrastructure: A technology which leaves the responsibility of choosing the programming language to the developer

```
<UserControl x:Class="HelloWorld"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
 xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
 mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">
 <Grid x:Name="LayoutRoot">
  <Button Name="button" Click="Button_Click" Width="100" Height="30"
   Content="Click me" />
 </Grid>
</UserControl>
```

Listing 1.8: File »HelloWorld.xaml«

The following, to the HelloWorld.xaml-file connected, C# class catches the click-event on the button and shows a message-box to the user.

```
namespace HelloWorld{
 // Controller class for HelloWorld.xaml (extends UserControl)
 public partial class HelloWorld : UserControl{
  public HelloWorld(){
   InitializeComponent();
  }

  // Event listener. Called if the button was clicked
  private void Button_Click(object sender, RoutedEventArgs e){
   MessageBox.Show("Button clicked");
  }
 }
}
```

Listing 1.9: File »HelloWorld.cs«

Silverlight is optimized for Windows operating systems. There is a version for Mac OS X as well but none for Linux-based systems. Moonlight, an open source 3rd party framework from the Mono project[14], offers a runtime for Silverlight applications on Linux, which is unfortunately often 1-2 versions behind the official one from Microsoft.

1.3.4 JavaScript

JavaScript, also simply called JS, is an object-oriented scripting language, which does not belong to a specific company but was founded and first implemented by Netscape back in 1995. It's core, the ECMAScript, is standardized by the ECMA organization and is thus an official web standard. Despite other RIA technologies like Flash or Silverlight,

[14] http://www.mono-project.com/Moonlight

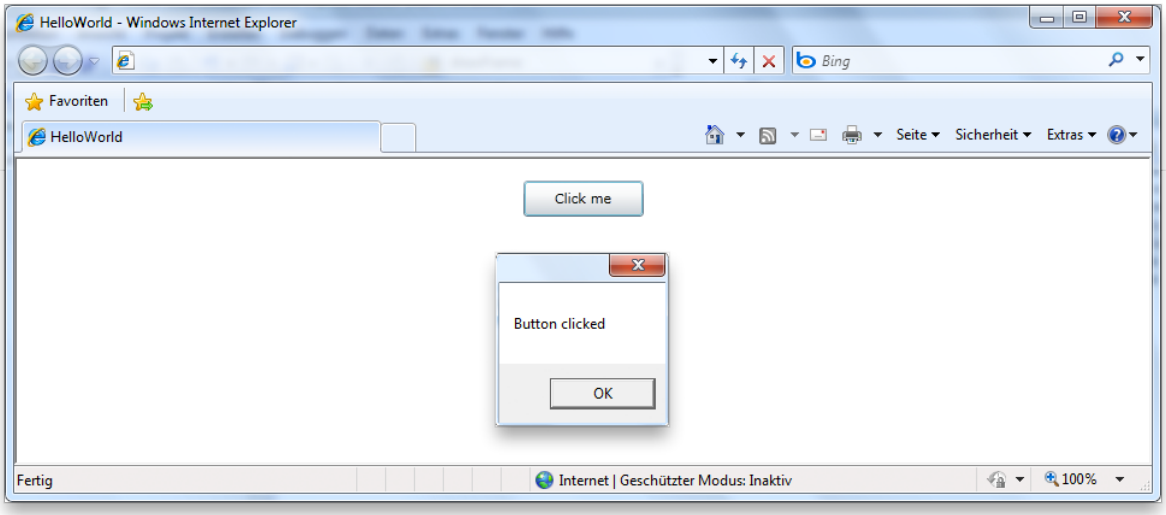

Figure 1.4: The compiled hello-world example application for Silverlight

JavaScript does not require a plug-in. Web-browsers have their own JavaScript engines, which sometimes lead to incompatibilities and unexpected behavior in early days of this language. Today, JS is absolutely essential for so called Web 2.0 applications, which heavily rely on dynamic content, processed by the browser's own scripting engine on the user's computer. As already mentioned, JavaScript is an object-oriented language but it does not implement the concept of classes or interfaces. Instead, object-orientation is being accomplished by a combination of anonymous functions and prototypes, which leads to unusual syntax if compared to traditional oo[15]-languages like Java. The example below shows how this concept works in JS.

Listing 1.10 on page 23 shows the definition of a class called »ClassA«. Since JavaScript does not know the `class`-keyword, polymorphy is being implemented by using functions, which allow encapsulating other functions. The constructor receives two values `parameter1` and `parameter2` and must be called explicitly (see line 20). Three lines above, the declaration of the class properties is being done. Note the differences regarding syntax between a private (defined through keyword »var«) and a public attribute (keyword »this«).

Visibility is a general problem in JavaScript, as illustrated in line 7. It is not possible to directly access public properties from within private methods. In order to accomplish this, an extra variable (called »selfPointer« here), which holds a reference to the instance of this class, is required. Using »this« instead of »selfPointer« would **not** work.

[15] Object-oriented

```
1  // Class definiton
2  var ClassA = function(parameter1, parameter2){
3
4   // Constructor
5   var constructor = function (parameter1, parameter2){
6    privateProperty = parameter1;
7    selfPointer.publicProperty = parameter2;
8   }
9
10  // A public method for getting a private property
11  this.getPrivateProperty = function(){
12   return privateProperty;
13  }
14
15  // Class properties
16  var privateProperty = null;
17  this.publicProperty = null;
18  var selfPointer = this;
19
20  constructor(parameter1, parameter2); // Call the constructor
21 }
```

Listing 1.10: File »ClassA.js«

```
1  // ClassB which should extend ClassA
2  this.ClassB = function(parameter1, parameter2, parameter3){
3
4   // Call the super-constructor
5   this.constructor(parameter1, parameter2);
6
7   // Add a new property
8   this.newProperty = parameter3;
9  }
10 // Let class B extend class A using the prototype property
11 ClassB.prototype = new ClassA();
```

Listing 1.11: File »ClassB.js« (extends class A)

In order to complete the example, a second JavaScript class (»classB«) is being defined in listing 1.11 on page 23, which extends the priorly defined »classA« by adding a third parameter to the constructor as well as declaring a new public class property called »newProperty«. Since JavaScript does not know the keyword »extends«, which is often used in object-oriented programming-languages like Java or ActionScript, polymorphy is being accomplished through the so called »prototype-concept«: Each object in JavaScript owns a public property called »prototype«, which can be used to let `classB` extend `classA` by simply assigning an instance of classA to it (see line 11 in the source of listing 1.11).

ClassB can now be used like this:

```
var classB = new ClassB(1, 2, 3);
alert(classB.getPrivateProperty()
 + " " + classB.publicProperty
 + " " + classB.newProperty
);
```

The code above will show a message box, with the following content: `1 2 3`

Developers with a C++ or Java background might be irritated by the prior example but that's the way object-oriented programming works in JavaScript, no matter how confusing the syntax might be. Besides this issue and the lack of interfaces as well as the concept of abstract classes, a huge problem in JavaScript is that there are no static types and no compiler. This sometimes leads to unstable programs due to type incompatibility issues, which were not noted before runtime execution, since JavaScript code is only being interpreted but not compiled. Thus, static type checking is not possible. For example, the following code is valid in JavaScript:

```
var test = 1.3;
alert(test);
test = "This is a string";
alert(test);
```

This snippet will be interpreted and executed by any JavaScript engine without errors.

1.4 Motivation and personal experience

By the time this thesis was written, huge discussions around Flash technology were going around, caused by the decision of computer-manufacturer Apple Inc. to not allow Flash applications on their products iPod Touch, iPhone and iPad. Also, an open letter[16] by Apple's CEO Steve Jobs lead to even heavier debates around Adobe's RIA technology due to his accusations on Flash being unstable and slow. Although it is often claimed that performance on Flash-applications is not very good, there is no scientifically approved evidence backing up these statements. Thus, it was interesting to see if there is any truth about Job's accusations regarding Flash-performance in his letter.

A second intention of this thesis is to verify if the rumors, which are going around on blogs and forums on the World Wide Web, that performance on plug-in-based technologies (like e.g. Adobe's Flash) are being influenced by the web-browser they run in. Similar to Steve Job's accusations on Flash application performance, these claims are not backed up by any (scientific) evidence and thus require detailed analysis.

[16] See `http://www.apple.com/hotnews/thoughts-on-flash/` for the full letter

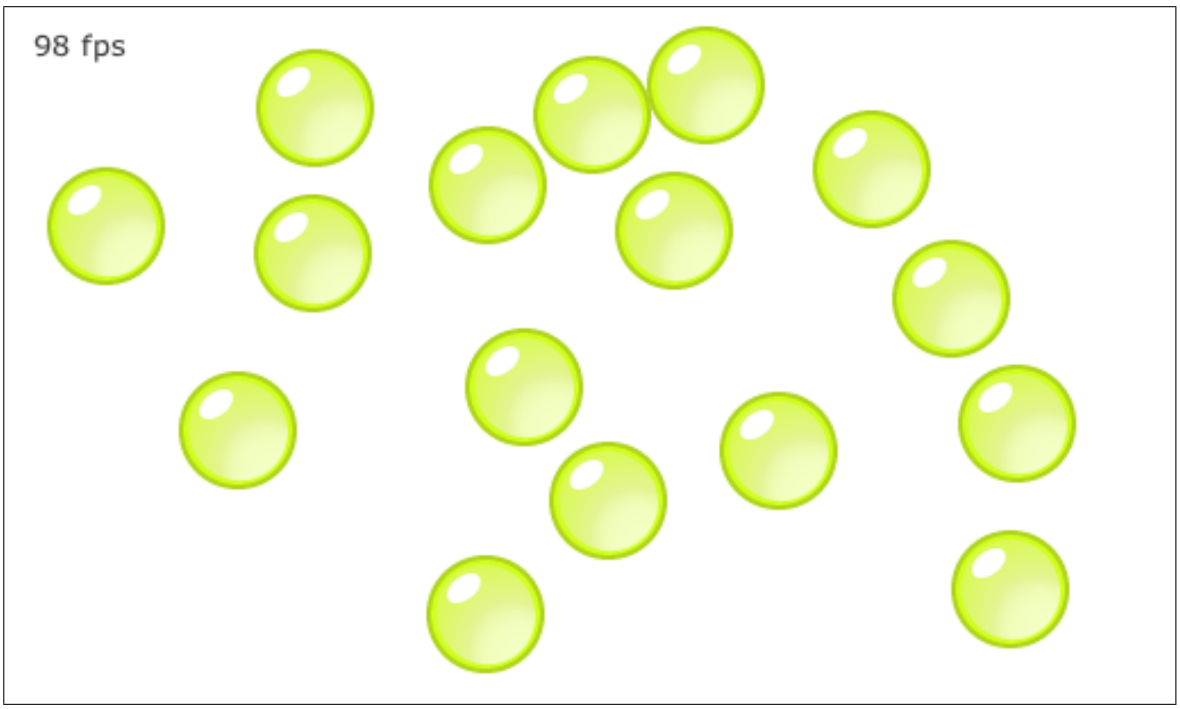

Figure 1.5: Bubblemark benchmark by Alexey Gavrilov

Personal experience

Since the author of this thesis started developing Rich Internet Applications using Adobe's Flex SDK, it happened quite often that performance issues slowed down the development process of the product. While the creation of GUI elements itself was easy, performance-optimization was often a »true nightmare«.

During an internship at IBM Germany back in 2009, the author had to develop a Visualizer based on Flex that heavily relied on its charting library API. Even on strong machines, it was not possible to create more than 20 charts on one screen at the same time. If tried, the application terminated with a timeout exception after 60 seconds because it simply took the rendering engine to long to draw all the charts at once. These experiences lead to thoughts about questions why the Flash Player sometimes performs so slowly and if other technologies like JavaFX or Silverlight could do any better. While looking for answers, the author encountered two benchmarks. One is Alexey Gavrilov's Bubblemark test[Gav] (See fig. 1.5 on page 25) which moves around bitmaps on the screen capturing the current fps. The other one is Sean Christmann's GUIMark[Chr] (See fig. 1.6 on page 26), which simulates a common website layout and lets it scale up and down. While Gavrilov's attempt is rather simple, Christmann's benchmark is a bit more complex including aspects like transparency and overlapping layers. Both tests include technologies like Flash/Flex, JavaFX, Silverlight and Javascript. All these attempts have one thing in common though: They represent only one big benchmark

Figure 1.6: Screenshot of Sean Christmann's GUIMark benchmark

instead of cutting down the issue into multiple aspects. This leads to the problem that one cannot clearly see what the reason is why solution A is faster or slower than B.

For example: Moving around bitmaps, as shown in Gavrilov's Bubblemark benchmark, may sound simple but heavily relies on multiple aspects of a RIA runtime: First, to display images, a graphic-buffer needs to be filled with the bitmap data. Then it needs to be drawn to a canvas-like component and finally shown on the screen. To move around the images, mathematical calculations are required to let the balls bounce from the walls. Furthermore, some kind of data structure like (dynamic) lists or arrays must be used in order store each ball-object in. While running the test, one never knows what was the cause for performance decreases. Was it the »physics engine«, the image processing calls, the array/list operations or something else? This lead to the idea of developing a *series* of tests to drill down to the core of performance issues, which leads to two benefits: One is that developers who already know their requirements for their applications can choose the RIA technology that fits best for their needs, based on the result of these test series. The other one is that RIA manufacturers can optimize their virtual machines and browser plug-ins based on the conclusions of this thesis.

2 Performance experiments

»Someone once told me that time was a predator that stalked us all our lifes, but I rather believe that time is a companion who goes with us on the journey and reminds us to cherish every moment because they'll never come again.«

(From the movie »Star Trek Generations«, 1994, Paramount Pictures)

2.1 Preparation

Unlike Captain Picard from the space ship »Enterprise«, human beings often see time as an enemy. People are in a hurry, every day. They need to catch the bus in time, meet with a date or got an important meeting. Sitting in front of a computer waiting for an application to finish some procedure is a waste of time. Thus, performance plays an important role in today's computer requirements regarding both, hardware as well as software. Therefore, it is important to know where the strengths and weaknesses of certain technologies are. In this case, four RIA runtimes will be examined regarding their behavior in various situations. The machine all tests run on is a Apple Macbook Pro using an Intel Core2Duo processor running at 2,53GHz with 8 GB of RAM. Beside the already preinstalled MacOS X (Snow Leopard), Bootcamp was used to setup Microsoft Windows. To keep things simple, Ubuntu was installed using Wubi, which enables one to install Linux on a NTFS filesystem inside one large container file. The benefit of this technique is that one has not to worry about issues regarding hard disk-partitioning or boot loaders. The only drawback is a little performance decrease if it comes up to filesystem access using Linux. This is tolerable since all tests rely on CPU power rather than hard disk speed.

Regarding the software-side of the tests, there are basically two types of RIA platforms, which would be those that require additional external software installation, like virtual machines or browser plug-ins, and those which work without any kind of extras.

The following lists all technologies used in this thesis:

- **JavaScript**
  Runs in various browsers. No extra software required.

- **Adobe Flash with Flex 3.2 framework**
  Tested in Flash Player 10.0 and 10.1. Browser-plug-in required.

- **JavaFX 1.3**
  Runs in Java Virtual Machine (version 1.6.0). Browser-plug-in required.

- **Silverlight 3**
  Runs in Silverlight 4.0 Runtime Environment on Windows and Mac OS. For Linux, the 3rd-party implementation called »Moonlight« (ver. 3.0 beta) from the Mono Project was used. Browser-plug-in required in both cases.

All tests are being run on these browsers:

- Opera 10.10

- Opera 10.54

- Apple Safari 5.0

- Google Chrome 5.0

- Mozilla Firefox 3.6

- Internet Explorer 8

All tests are being run on three operating systems, which are:

- Windows Vista (32 Bit, Business Edition)

- MacOS X 10.6 (64 Bit, Snow Leopard)

- Linux (32 Bit, Ubuntu 10.04)

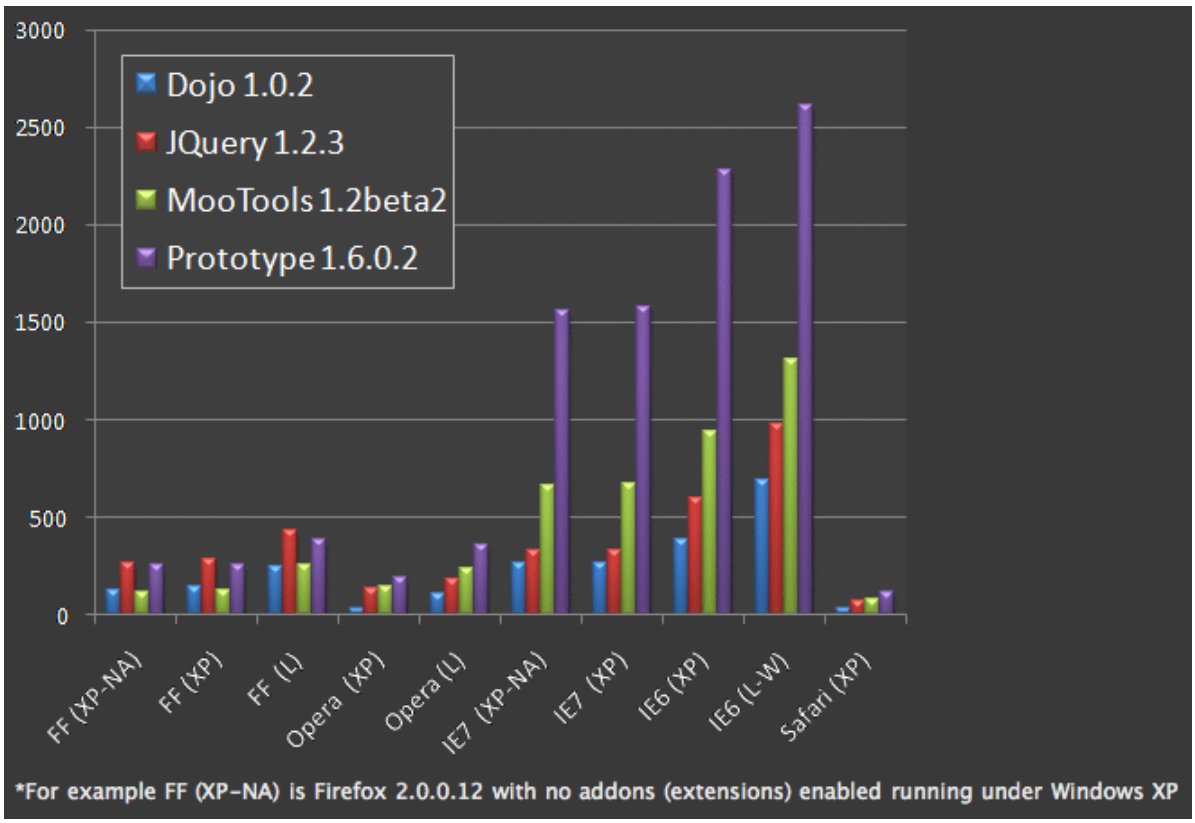

Figure 2.1: JavaScript framework benchmark results by Peter Velichkov

Notes

- The benchmarks and their source code can be downloaded online from: *http://www.timo-ernst.net/riabench-start*

- Because of platform dependence, tests on the Internet Explorer are only possible on Microsoft Windows.

- On Linux, there is currently no Safari and no Opera (version 10.54) browser available, which is the reason why there are no such test for Ubuntu.

- The fact that the 32-Bit versions of Windows and Linux in these tests cannot address the full 8 GB of RAM does not affect any measurement results since none of the tests require more than a few kilobytes of memory.

- Since JavaScript is the only RIA technology that does not offer a build-in api, for some tests, JQuery 1.4.1, a very common framework, was used. There are three reasons why:

1. Cross-browser issues can lead to problems developing JavaScript applications. Frameworks like JQuery take care about these problems using best-practice techniques.

2. Tests from Peter Velichkov[Vel08] (See fig. 2.1 on page 29) using Slickspeed[Moo], a performance measurement tool for JavaScript frameworks, have shown that JQuery is one of the fastest JavaScript libraries currently available.

3. JQuery is widely spread among web-developers and thus often used which makes the results of these series useful in real projects.

- In order to keep source code(-fragments) printed in this thesis as small as possible, all `import`- and packaging statements were removed.

- Direct comparisons between operating systems will not be made here because their architectures, especially regarding memory management, are so different which makes it very difficult to achieve objective results. Thus, only relational comparisons are being made, like for example: »Flash was three times faster than JavaFX on Mac OS X. The same could be observed on Microsoft Windows.« But there will never be anything like: »Flash on Mac OS X was three times faster than on Windows«.

- Regarding licensing: Most of the used code here in this thesis is licensed either under the GPL or MIT. Because of copyleft issues, not all tests use the same license. Detailed information regarding this can be seen inside the provided source code for each test.

- All implementations, no matter on which platform, will start 1 second after they were called. The reason for this is that some kind of notification to the user (that the test has started) must be displayed in order to give feedback that the application has not frozen. Without the delay, the test would be immediately started without printing anything to the screen, even if the lines containing the output commands are above the testing parts in the source code. As an example, here is the JavaScript version of the delay, which is similarly implemented in all other RIA solutions:

```
// Will be called on body.load() event
function bodyLoaded(){
 setTimeout("startTest()", 1000);
}
```

- All test-cases have the following structure in order to maintain a good overview:

  1. Motivation—The reason why this test was included into the series.

  2. Test setup—A short explanation about how the test was planned and implemented.

  3. Results—What worked and what didn't? How long did each test take?

- Regarding the visualization of the results for each test though charts, it must be said that due to limitation of space only the interesting ones are printed into this thesis. See the attachments on page **??** and later for the tables containing all results in detail and `http://www.timo-ernst.net/riabench-start` for all charts to download.

- Due to limitations of space in this thesis, not all results, as well as charts visualizing these, could be printed here. For a full list of all measurement values, see the attached CD-ROM or visit `http://www.timo-ernst.net/riabench-start`

Caching

During the development of these tests, it could be observed that some RIA platforms seem to cache information in order to boost application's running speed next time they're started. Figure 2.2 on page 32 shows the results of a short test where a JPEG-encoding algorithm was used to compress a sample png image (see section 2.3.1.1 on page 36 for detailed description and analysis on this benchmark). Test browser was Apple Safari running on Mac OS X 10.6.3 (Snow Leopard). In order to be able to compare cached vs non-cached results, all caching systems on browsers and RIA runtime environments were disabled and existing data deleted using the methods explained below. Then, all tests were run again with caching enabled.

- **JavaScript**

  All browsers used in this thesis provide full control over their caching system, which includes enabling, disabling and cleaning (=removing) its content. For example[1]: In Google Chrome, all cached data can be deleted by opening the browser's menu and selecting:

  Chrome → Delete internet-data → Empty Cache → Delete internet-data

[1] This example only applies for Google Chrome on Mac OS X. Other operating systems and browsers provide similar options, hence the presentation of these were omitted here.

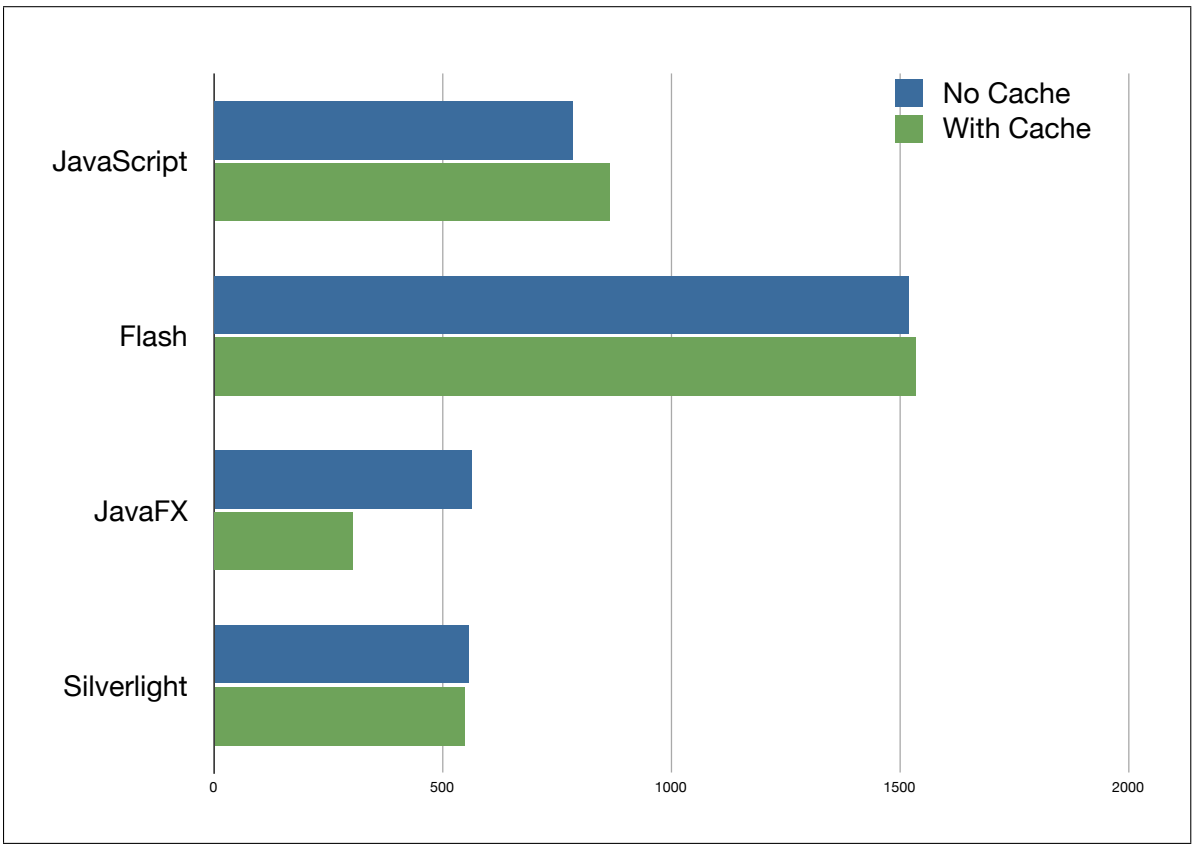

Figure 2.2: Results of the JPEG encoder benchmark with and without caching (MacOS)

Thus, whenever caching was not desired, both options (disabling **and** cleaning) were activated on all browsers before any tests were run.

- **Flash**

  The Flash platform does not offer a built-in configuration manager on the local operating system in order to modify its caching behavior. Instead, Adobe provides a web-application, written in Flash, which allows the user to do these configurations online at:

  `http://www.macromedia.com/support/documentation/de/flashplayer/help/settings_manager.html`

  In this application, caching was disabled by opening the tab »Global memory settings« and reducing the allowed storage capacity to 0 as well as deactivating both check-boxes on the bottom. Furthermore, in the tab »Website memory-settings« all existing cached data was deleted.

- **JavaFX**

  Since JavaFX applications run inside the Java Virtual Machine, all settings regarding Java Applets also apply for JavaFX. In order to modify these options, Java offers a settings-manager which allows the user to manipulate caching-behavior under the tab »Network«. For the purpose of this thesis, two settings were made here:

  1. Disable the checkbox »Keep temporary data for quick access«
  2. a) Click »Remove files...«
     b) Select option »Programs and Applets«
     c) Click OK

- **Silverlight**

  Silverlight does not have an own caching system. Instead, all caching needs are being delegated to the web-browser the application runs in. Thus, the priorly mentioned conditions for JavaScript apply.

As it can be seen in figure 2.2 on page 32 only JavaFX (∼50% speed increase) and JavaScript (∼10% increase) benefit from their caching system. Flash and Silverlight do not show a big difference whether caching was enabled or not:

Caching option	JavaScript	Flash	JavaFX	Silverlight
Off	785 ms	1519 ms	563 ms	556 ms
On	864 ms	1533 ms	304 ms	548 ms

The small differences between the cached and non-cached versions for Flash and Silverlight are probably caused by background tasks running on the local operating system. This assumption can be backed up by simply re-loading the same application over and over again. The results always fluctuate by about 1-2% which can surely be ignored.

Conclusion: Caching does definitely affect performance for JavaFX as well as JavaScript applications while Flash and Silverlight do not benefit from their cache-systems. In order to get comparable results in all benchmarks achieved in this thesis, **all tests** will be run with **caching disabled**, no matter which RIA runtime is being used.

2.2 Test strategy

While implementing the test cases, there were two important aspects, which significantly influenced the development process, which were:

1. Practical relevance
2. Test type categorization into:
    - Use-case tests
        - API-tests
        - Non-API tests
    - Focus tests

The basic idea behind this »strategy« is to first implement common RIA-related use-cases, like for example image compressing, and examine their runtime-behavior using debugger-tools, like the built-in tools for Eclipse, in order to search for potential performance bottlenecks, conspicuous behaviors between the RIA-technologies and anything else obtrusive.

Note: Due to limitations of space in this thesis, not all results, as well as charts visualizing these, could be printed here. For a full list of all measurement values, see the attached CD-ROM or visit `http://www.timo-ernst.net/riabench-start`

2.3 Use-case tests

The Use-case tests are separated into so called Non-API-tests (which use functions and classes of existing libraries) and API-tests (which are built on top of self-written algorithms whenever possible). There are several reasons for this categorization:

1. Performance issues can either be caused by the RIA platform itself (the virtual machine or browser plug-in for instance) or by the implementation code inside the API. For example: Imagine a simple 3D scenario where a cube is being displayed on the screen and rotated around its own axis. To visualize this, you need a 3D engine and a platform where the application runs on. If the virtual machine is fast, there is still a possibility left that the rotating animation won't be smooth enough. This can happen if the API implementation, in this case the 3D engine, is

implemented inefficiently. Now, if the opposite happens (slow virtual machine and fast implementation) the result can also be dissatisfying.

2. From the perspective of a RIA developer, there is no reason why one should write an own implementation if there is already a working one in the provided API, which is the reason for aspect number 1 (»Practical relevance«). It does not make sense, testing a RIA platform with an own implementation of a 3D engine, if 99% of the »real-world-developers« will use an existing framework.

3. The strategy in this thesis for finding performance bottlenecks is to first write tests in a fairly wide scope by picking some web-relevant use-cases and then, based on these test results, try to focus on potential bottlenecks. Example: In this thesis it will be shown that the run-length encoding test, which heavily relies on string-operations (e.g. concatenation etc.), runs very slow on some browsers, like for example Mozilla's Firefox. Based on this insight, a new, lower-level test was developed for string-operations to have a deeper look into this issue.

Thus, all three aspects had to be taken into consideration when these performance tests were developed in order to find the real cause for slow application behavior. Based on the results of these Use-case tests, all the conspicuous aspects will be taken out and explicitly tested in one separate focus-test, called *RIABench*.

2.3.1 API tests

API-tests were created by using as many existing implementations of required algorithms as possible, which also includes external frameworks as well as code snippets (and not only the build-in API-functionality of the underlying RIA platform). The reason for this is that most developers usually first search for existing implementations and then, if nothing was found or a license mismatch occurred, own code will be written. This way of developing software has become quite popular. On one hand, the benefit is clearly visible: Development can become faster since code won't have to be re-coded if there is already an existing implementation. On the other hand, one cannot clearly see how robust and effective the source is, especially if it has a lot of lines.

However, for the purpose of this thesis the following test cases were developed using as much existing code as possible.

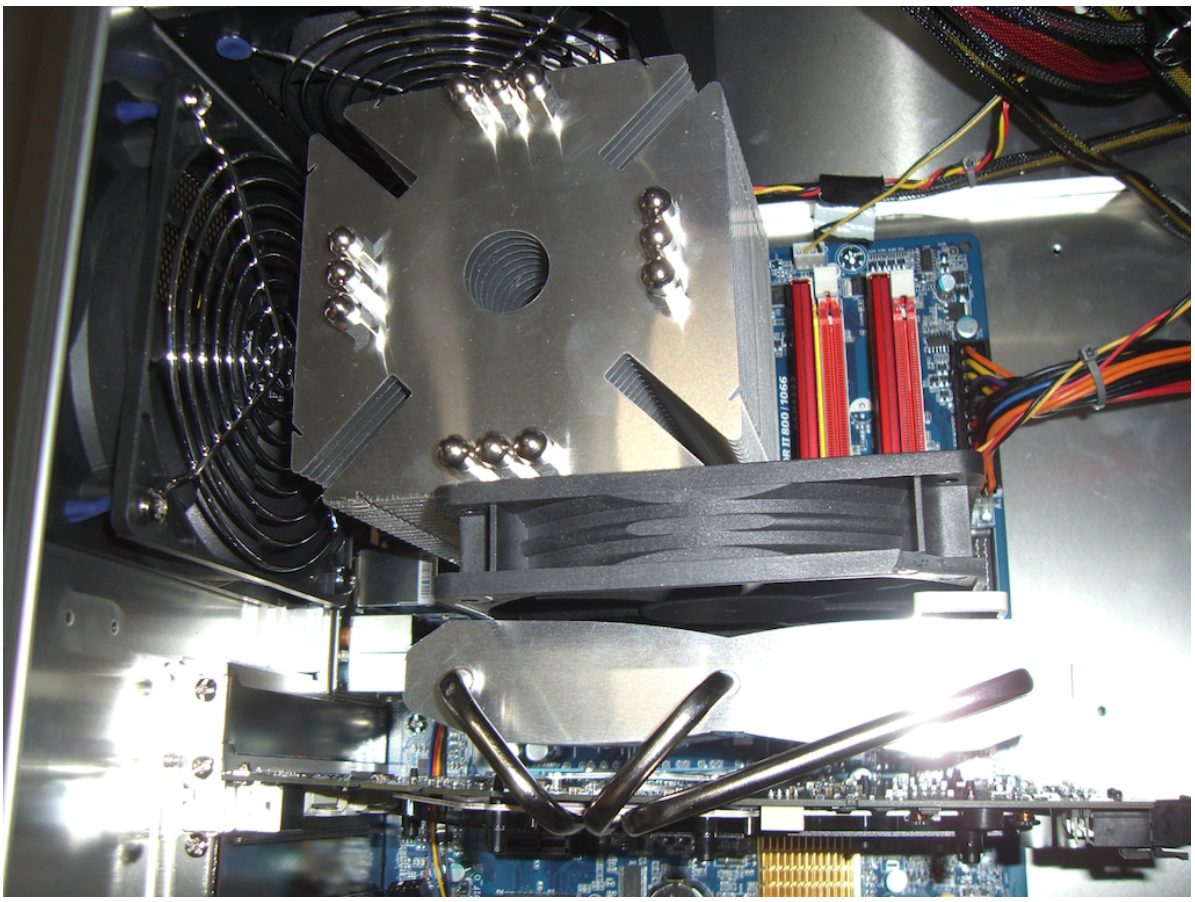

Figure 2.3: JPEG encoding test image 1

2.3.1.1 JPEG encoding

Image processing itself has become important since the rise of the so-called »Web 2.0«. Users upload pictures (and other media) to their weblogs and social networking sites like *Blogger* or *Facebook*. Thus, the ability of resizing and compressing images has become more important than it was before. Especially the ability to do these kind of operations on the client itself (e.g. in the browser) is tempting since it can reduce the number of requests to the backend system and thus, minimize the load on its servers. The benefits are clearly visible: Smaller images imply shorter loading times for the visitor of a website while huge pictures could even break a page's layout, if not explicitly limited.

Test setup

The testing strategy is basically the same for all RIA runtimes: One picture, showing the inside of a pc (See fig. 2.3 on page 36), should be compressed using the JPEG algorithm. Source file format was PNG and the dimension 1024x768 in sRGB. On a scale from 0

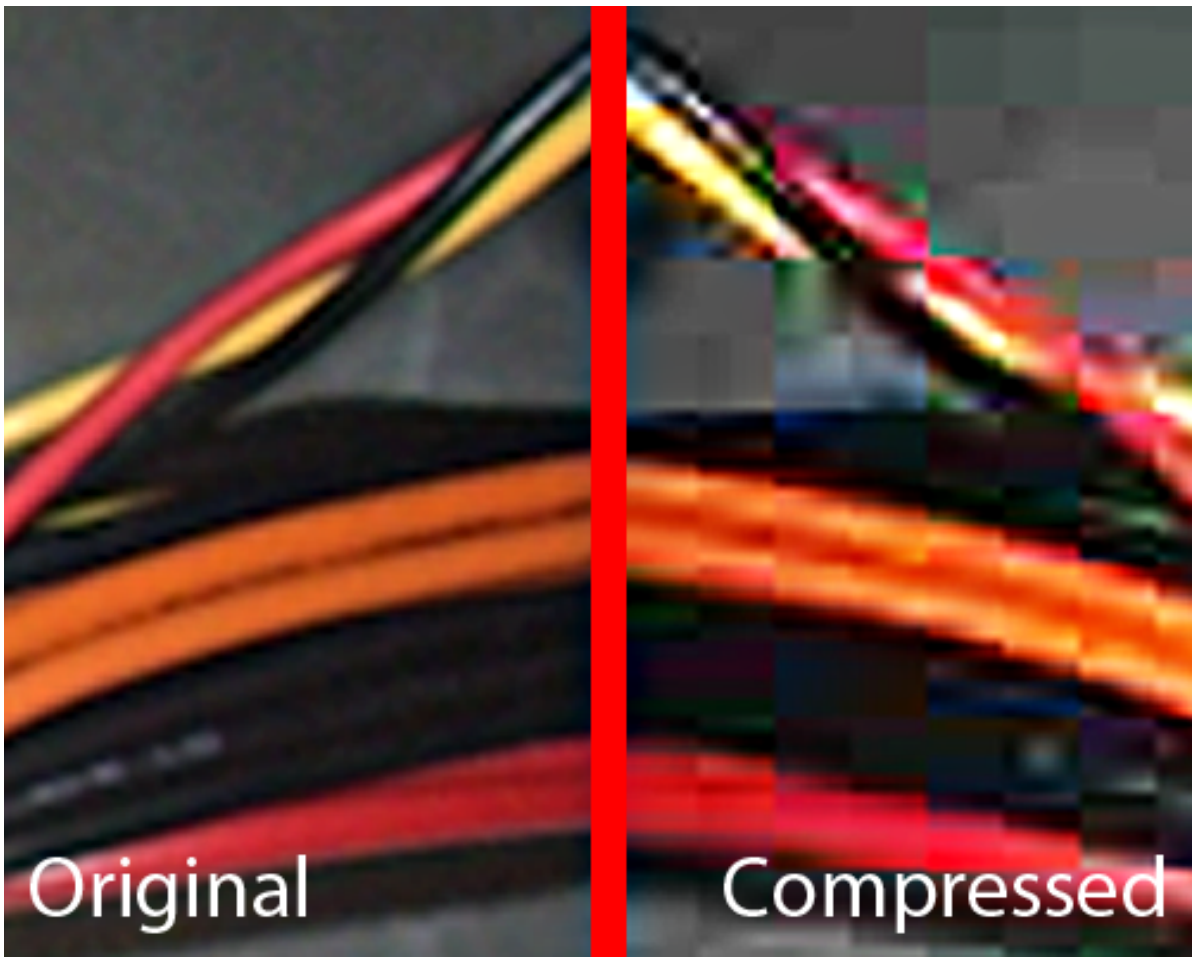

Figure 2.4: Partial magnification of test image 1 before and after JPEG-compression with visible aftefacts

to 100, where 0 equals worst and 100 best quality, the setting for this test was 20. This means that the compressed image will have quite bad quality, but for the purpose of this test, the resulting visible artefacts can be a simple but effective indicator whether the encoding process was successful or not (See fig. 2.4 on page 37).

The test-strategy itself is pretty simple:

1. Start timer
2. Encode image
3. Stop timer and calculate elapsed time

Since **JavaScript** does not offer any JPEG encoding functionality, a third-party implementation was used here. The code was provided by Andreas Ritter[Rit10], who ported

an ActionScript version of the algorithm, which was originally developed by Adobe Systems for the Flash platform, to JavaScript. Since both scripting languages share similarities regarding the syntax, this code snipped worked quite well and stable during the tests. The results should be very interesting since this is the only implementation of the JPEG algorithm which makes it possible to compare the JavaScript engine performance against the Flash Player without bothering about the algorithm implementation since the code for both should be almost equal (except for minor differences regarding syntax).

The **Flash** version of the test works quite similar: Since the API on the Flex framework offers a large variety of different libraries, it was no wonder that the previously mentioned JPEG encoder class was already built-in to the `mx.graphics.codec` package. Thus, it was very easy to implement the test: All that was needed was to import the image into the Flex project, send it through the encoder and check how long it would take.

JavaFX has one big advantage compared to other RIA solutions, which is the possibility to use almost everything of the whole standard Java API. This interface is already well-known among most software engineers and thus it is not necessary to rewrite code using JavaFX Script as long as an implementation for Java already exists. In this case there is already a JPEG encoder available which makes it easy to do the JPEG compression in pure Java, send back the result to a JavaFX front-end program and then display it. This architecture-style is typical for JavaFX applications: All the front-end logic is being done in JavaFX-Script while the »real magic« happens in ordinary Java: First, the input file is being read from the file system which is then encoded by a JPEG-ImageWriter. In order to be able to display the result in a JavaFX application, the compressed picture had to be converted into a BufferedImage object (see listing 5.2 on page 126).

Using this Java-encoder, it is possible to pass the path to an existing image to it, receive the compressed version and then display it on screen:

```
var imageView1 = ImageView {
 fitWidth: 1024
 fitHeight: 768
}
var img1:Image = Image{};
var encoder:JpegEncoder = new JpegEncoder();
var imgBuf1:BufferedImage = encoder.encode("image1.png");
// Convert the result to a JavaFX image object
img1 = javafx.ext.swing.SwingUtils.toFXImage(imgBuf1);
// Display the image
imageView1.image = img1;
```

Although the basic idea behind the **Silverlight** version was the same as already mentioned for JavaFX or Flex, the implementation itself was a bit more complicated since the API does not offer a build-in encoding function for JPEG. Thus, FJCore, a popular library for multimedia purposes in C#-based applications, was used.

Results

- Opera 10.10 performed worst on JavaScript on all operating systems while its successor (version 10.54) is about as fast as the best JavaScript-browsers Google Chrome 5.0 and Safari 5.0.

- Firefox 3.6 on JavaScript was about 3 times slower than the top browsers (Chrome, Safari and Opera 10.54) but still more than 2 times faster than Opera 10.10.

- JavaFX and Silverlight performed best on all operating systems including Moonlight with the top result on this benchmark of 424ms (see figure 2.5 40).

- Regarding Flash, only the Mac-version of 10.1 was noticeable faster (2.8 times) than its predecessor 10.0. On other operating systems, the speed increase was way less (factor 1.3 on Linux and 1.4 on Windows).

- Compared to other RIA solutions, it can be said that Flash performed worse than the top runtimes JavaFX and Silverlight/Moonlight as well as JavaScript on Chrome, Safari and Opera 10.54 but still outran Opera 10.10 and Firefox 3.6.

Additionally, it must be said that the JavaFX plugin often did not work on various combinations of browsers and operating systems. Opera in version 10.10 refused to run such applications on all platforms although its new version 10.54 included support for JavaFX. Other combinations of browsers and operating systems, which couldn't run JavaFX content were Google's Chrome on Mac OS X as well as Apple's Safari on Microsoft Windows Vista. Regarding the test on Microsoft's Internet Explorer 8.0, it must be said that the JavaScript-version of test could not be run due to the lack of HTML5's new `<canvas>` element in this browser, which is absolutely vital for the benchmark. The reason why the JPEG compression without `<canvas>` is not possible is that the bitmap-data from the original image must be read-in somehow in order to apply the JPEG algorithm on it. Since there is no such element in the Internet Explorer 8.0 browser, this test failed. As mentioned in the prior sub-section about the test-setup for this benchmark, the implementations for the JavaScript- and Flash-version should be very similar. Thus, it was interesting to see that the JavaScript-version performed a lot better than Flash on three browsers (Chrome, Safari and Opera 10.54) but in the same time, two browsers (Opera 10.10 and Firefox 3.6) were a lot slower than their ActionScript-pendants while the Internet Explorer variant refused to work at all. This leads to the conclusion that there is no real winner between both but generally it can be said that the JavaScript-versions are potentially better as long as the browser-engine is fast enough.

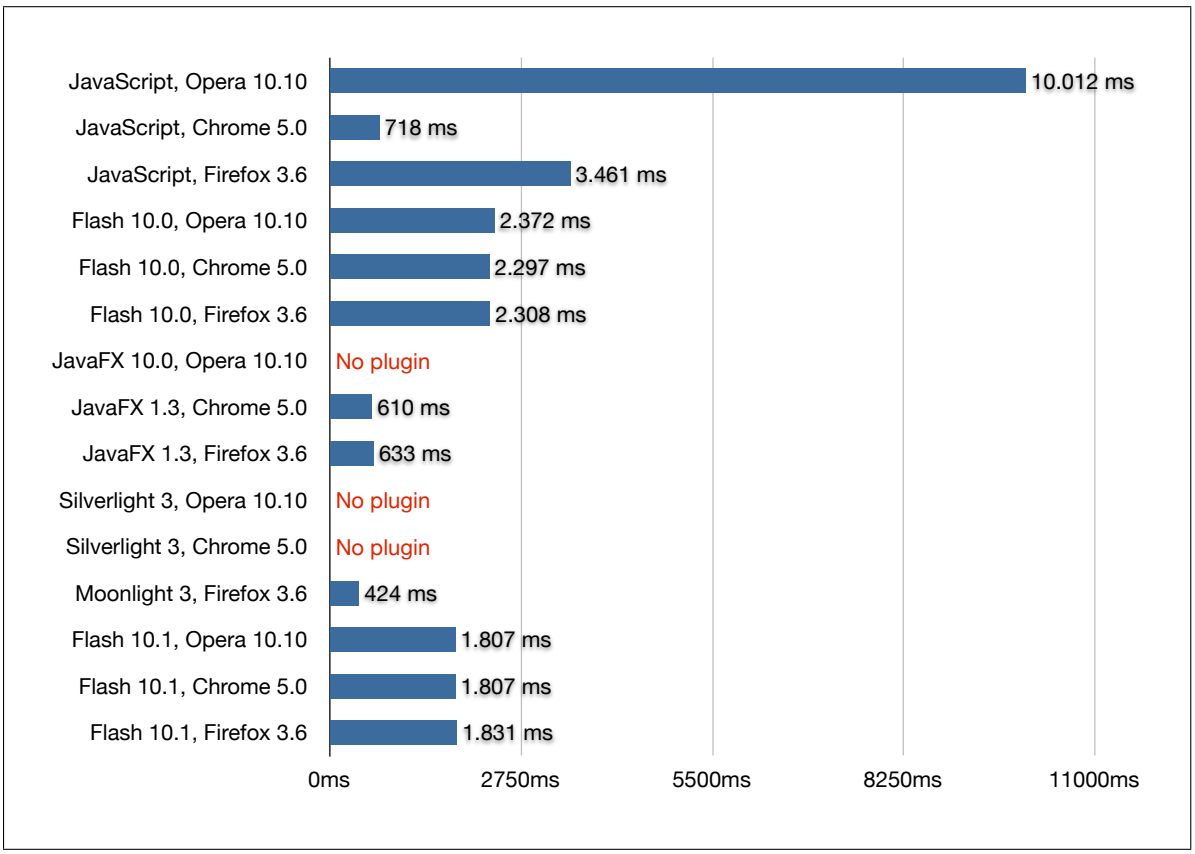

Figure 2.5: Best result for Moonlight (on Ubuntu Linux 10.04) across all operating systems for the JPEG-benchmark

2.3.1.2 MD5 hashing

The MD5[2] hashing algorithm was and still is important for the World Wide Web since it is often used for authentication purposes (e.g. if saving hashed versions of passwords in databases is a requirement), fingerprint testing (checksums) or signature verification (PGP[3]). There have been reports about found collisions in MD5, which makes the algorithm vulnerable, which again leads to the possibility of successfully creating two different documents with the same hash value, as described by Xiaoyun and Hongbo in »How to break MD5 and other hash functions«[WW05]. Although this implies a serious security threat, MD5 is still very present in today's web.

Just like most other cryptographic (hash-)functions, MD5 relies on the following operations [Riv92]:

[2]Message Digest, a popular hashing algorithm. See `http://www.ietf.org/rfc/rfc1321.txt`
[3]Pretty Good Privacy, an asymmetric en-/decryption as well as signature verification technology. See `http://www.ietf.org/rfc/rfc4880.txt`

- XOR, AND, OR
- Modulo
- Various array operations like pushing, selection, concatenating
- Bitwise left-rotation

The basic idea of this test is to read a given text file (455 kb size), hash it using the MD5 algorithm and see how much time passes for the encoding process to verify if any of the above mentioned operations could lead to performance fluctuations on specific RIA platforms. Based on the results, more detailed tests could be implemented (See section 2.3.2 on page 55).

Test setup

The implementation of the test for **JavaScript** was done by using JQuery with a special plug-in which supports MD5. Immediately before and after the encoder-call, the current timestamp was saved by simply instantiating a `new Date()` object, which saves the time in ms elapsed since January 1st, 1970:

```
var startTime = new Date();
var hash = $.md5(data); // JQuery call
var stopTime = new Date();
```

This way of time-measurement using (current) timestamps without timer-events has two benefits:

1. A timer could influence the result of a test since it requires some CPU time itself (even it is not much).
2. CPU-intensive tests could slow down the machine, so that the time measurement could be influenced and lead to wrong result data.

Thus, getting the elapsed time with the following line of code is probably the best:

```
// timeElapsed = Elapsed time between two timestamps in [ms]
var timeElapsed = stopTime.getTime() - startTime.getTime();
```

Surprisingly, the **Flex 3.2** API did not provide a built-in MD5 implementation, which made it necessary to use the external framework *as3corelib*, provided by Adobe itself.

As already mentioned in section 2.1, **JavaFX**, is based on a new scripting language, and »ordinary« Java. Since the standard Java Runtime Environment already provides

MD5 support through the `java.security.MessageDigest` package, it was decided to build the GUI using JavaFX Script and write the existing hashing functionality in pure Java:

```
import java.security.MessageDigest;
public class MD5 {
 public static String toMD5(String text){
  MessageDigest md = MessageDigest.getInstance("MD5");
  byte[] md5hash = new byte[32];
  md.update(text.getBytes("iso-8859-1"), 0, text.length());
  md5hash = md.digest();
  return convertToHex(md5hash);
 }
}
```

This architecture does make sense since no JavaFX developer in the world would probably start writing his own encoding algorithm if there is already a working implementation. Hence this strategy fulfills the requirement of »practical relevance« as described in section 2.2.

The implementation of the test for **Silverlight** is not much more different than compared to prior variants. Its API provides no built-in MD5 function, which made it necessary to use an extra class called MD5Core, written by Reid Borsuk and Jenny Zheng from Microsoft.

Results

On **Mac OS X**, the results show that both, JavaFX, as well as the Silverlight versions performed very well at rates between only 8ms (JavaFX, Opera 10.54) and a maximum of 54ms (JavaFX, Safari 5.0). Best Silverlight result was 16ms on Opera 10.54, Chrome 5.0 and Safari 5.0. Looking at the JavaScript-versions, it could be observed that Opera 10.10 (3174ms) as well as Firefox 3.6 (2586ms) represent the worst results while Opera 10.54, Chrome 5.0 and Safari 5.0 (average of 477ms) are pretty even with Flash 10.1 at an average rate of 289ms. Flash 10.0 performed almost 3 times slower than its successor at rates around 825ms - 849ms. Due to the lack of plug-ins, no results could be achieved on Opera 10.10 for JavaFX and Silverlight, as well as on Chrome 5.0 for JavaFX only.

Looking at the results from **Microsoft Windows Vista**, it is interesting to see that the distribution looks similar to the results from Mac OS. Besides Opera 10.10 and Firefox 3.6 with pretty bad values of 2335ms and 3237ms, the Internet Explorer 8.0 definitely returns the worst number at 4405ms. That is 629 times slower than the top value from JavaFX on Opera 10.54. The results on other browsers for JavaFX were almost equal (around 8ms - 32ms). No result could be achieved for JavaFX under Safari

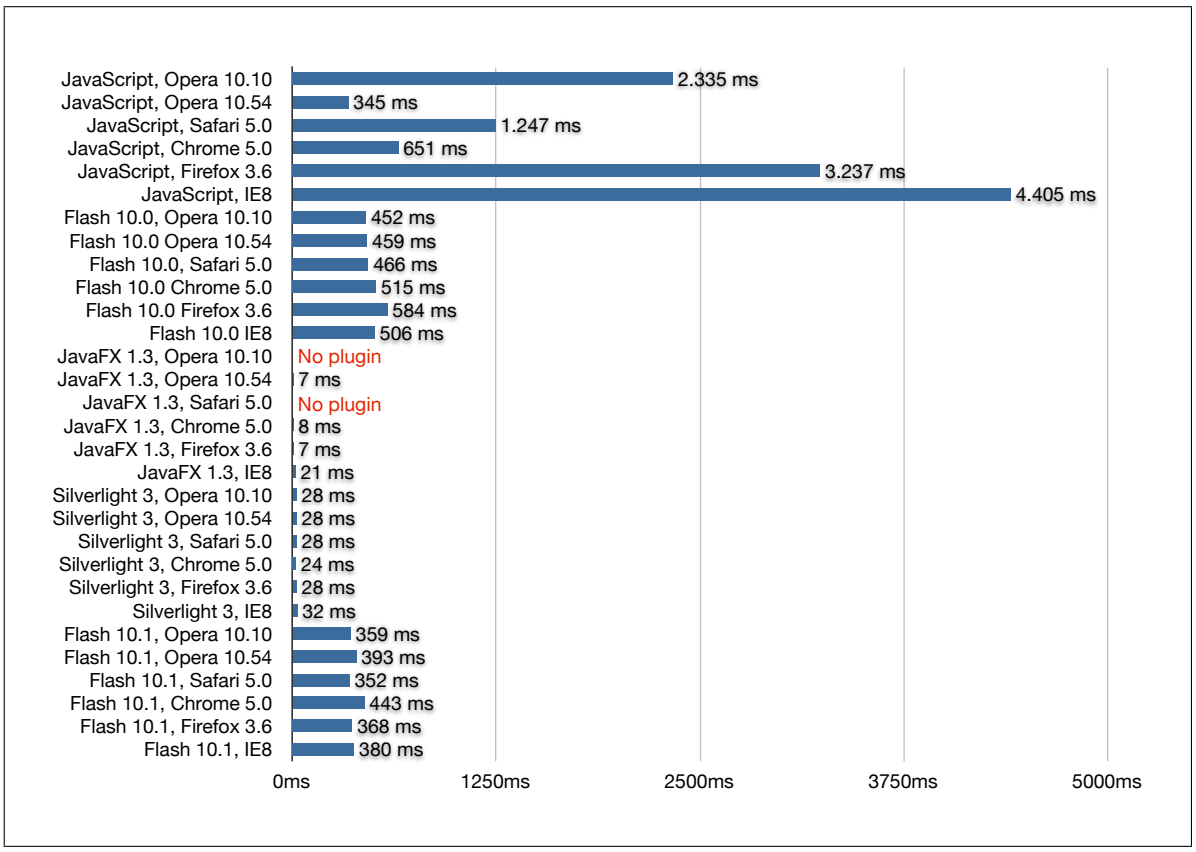

Figure 2.6: Results of the MD5-test on Microsoft Windows Vista

5.0 and Opera 10.10 due to the lack of an appropriate plug-in. While Flash 10.1 was only slightly faster than its predecessor version 10.1, the most astonishing result could be observed for the JavaScript-test on Apple's Safari 5.0. Although this browser uses the same Webkit-engine like Google's Chrome (651ms), its test result (1247ms) is about 2 times slower than the one of its competitor (see figure 2.6 on page 43).

Regarding the test-results on **Ubuntu 10.04**, it can be said that no in particular interesting observations could be made. Similar to results on Windows and Mac OS, JavaFX and Moonlight performed best while Opera 10.10 and Firefox 3.6 on JavaScript were the worst.

2.3.1.3 3D acceleration

The ability to play video-games on the web has become quite popular in the past years, especially on social-networking sites like Facebook (e.g. Farm Ville, see figure 2.7 on page 44 for an example screenshot). This made Flash *the* gaming development environment on

Figure 2.7: FarmVille: The browser as a gaming platform

the WWW[4] but other RIA platforms are already catching up, like for example Java(FX) with its built-in Java3D and JavaScript's WebGL which is being supported in some nightly builds of latest browser releases. Thus, it is important to be able to not only create games which are restricted to two dimensions. Developers want to use (hardware-) accelerated 3D engines in order to bring the whole world of desktop-gaming to the web. But it is not about gaming only: 3D-support could be useful for various kinds of Rich Internet Applications, like for example modeling-software for architectures/engineers to create 3D-prototypes of houses or cars for demo purposes.

Test setup

The basic idea of this test is to place some planets in an imaginary solar system rotating in a circular motion around one star (the sun). This setup scales very well since the number of planets is variable (and so is the complexity of transformations). The placement of the objects was done by using the pseudo random key generator (with different initialization values) from section 2.3.2.3 on page 62, which has three advantages:

[4] World Wide Web

1. Since the algorithm is self-written and tested, no unexpected behavior of the generator should influence the test results.

2. The same algorithm for generating random numbers was used on all RIA platforms (JavaFX, Silverlight, Flex, JavaScript), which makes the test results independent from different generator algorithms and their implementations.

3. The random key generator always produces the same numbers, based on the initial value (=the salt), which is 0 for all test implementations. If the salt is changed, so will the generated keys. Thus, all test-runs are 100% reproducible and thus comparable.

The usage of different initialization constants ($m = 9643$, $b = 232$, $a = 624$), as described in section 2.3.2.3, was made in order to archive smaller values. This may lead to worse numbers regarding the »randomness« but for the purpose of placing objects in a 3D-environment, this change will not lead to any issues. The generator creates a triple of integer values, which represent the coordinate c_i for the planet $p_i \in P$, where P denotes the set of all planets in this solar system.

$$c_i = \begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} \text{ with } \quad i, x, y, z \in \mathbb{N}_0$$

Furthermore, each planet p_i gets one of four textures $t_j \in T = \{t_0, t_1, t_2, t_3\}$, $j \in [0, 3]$ with $j := j \mod 4$:

- t_0 = »Earth texture«
- t_1 = »Moon texture«
- t_2 = »Mars texture«
- t_3 = »Jupiter texture«

(Sun excluded. See figure 2.8 on page 46 for the textures).

Since the number of textures $|T|$ for all tests with $|P| > |T|$ won't be enough, the allocation of textures for each planet is being calculated as $f(p_i) = i \mod 4 = j \; \forall p_i \in P$ with $f : P \to T$.

Example: Planet p_5 gets the texture for the moon since $5 \mod 4 = 1$ and thus $f(p_5) = t_1$.

Now, after all planets were placed, these spheres start rotating around the sun, whereas

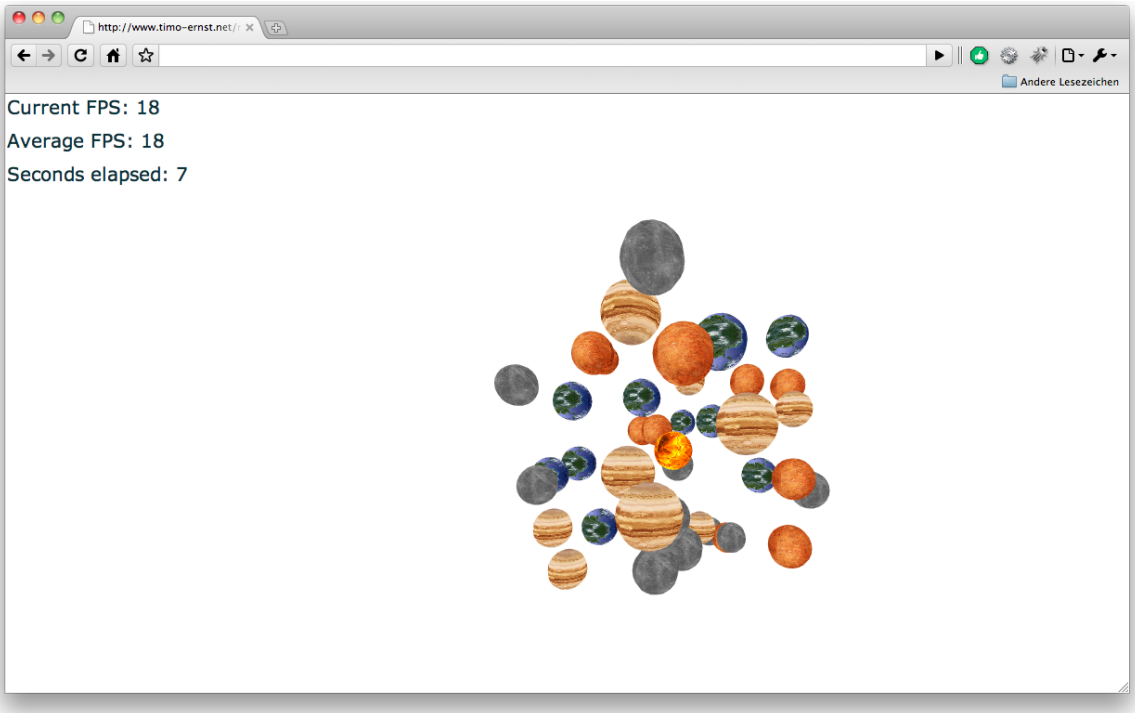

Figure 2.8: Screenshot of the Flash-3D-test in Google Chrome under Mac OS X

the solar system itself also has the shape of a sphere (=the geometric body with the biggest volume of all 3D-objects) and can therefore hold a maximum number of planets. Although there were no collision checks implemented in this test, this fact will at least reduce the potential number of collisions.

The animation speed is now being influenced by the following parameters:

- **Grade of detail (of each planet)**

  Usually spheres are being constructed by placing triangle objects together (see image 2.9 on page 48). The more triangles, the more detailed and thus rounder the globe will look. Therefore, it's obvious that the grade of detail leads to higher requirements regarding CPU/GPU power.

- **Number of planets**

  As mentioned before, each sphere is constructed through a group of triangles. Thus, it is clear that the number of triangles increases with the number of planets, which again leads to a potential decrease of the fps[5] rate.

[5]Frames per second

- **Planet-size**

  Some platforms automatically increase or decrease the grade of detail of certain objects depending on their size. Smaller objects usually need a lower grade of detail, since these are often further away from the viewers perspective while bigger (closer) objects require more polygons.

- **Rotation**

  For each animation step, a rotation animation around the center of the solar system must be done, which requires some matrix multiplication with the following vectors/matrixes:

  $$p_j = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

  $$M = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$$

  $$p'_j = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

  In order to rotate object p_j, the matrix M must be multiplied from left (since matrix-multiplication is not commutative) in order to rotate it to its destination point p'_j:

  $$M \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}$$

  M must be an orthogonal matrix with $det(M) = 1$. If not, the transformation will not lead to a rotation. The values a-i of matrix M depend on the desired destination coordinate.

- **Texture scaling**

  Since all planets must be displayed in different sizes, depending on their current location (e.g. near planets appear bigger), their textures must be scaled up or down.

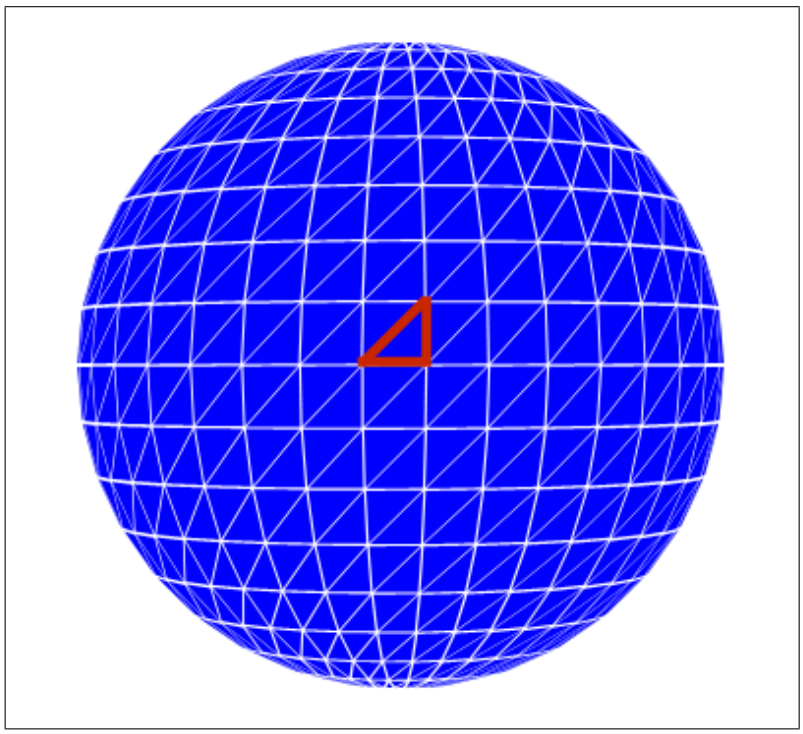

Figure 2.9: Sphere demo by Mark Dawson[Daw] built on triangles (marked red here).

In order to get a smooth animation, a frame-rate of 60 fps[6] was the goal for all test implementations. This was done due to the fact that most LCD screens do not refresh at higher rates than 60 Hz[7], as described in Sean Christman's article about »GUIMark Benchmark and Rendering Engine theory«[Chr]:

»*Since mainstream LCDs are fixed at 60 hertz, most applications have no need to render above 60 frames per second and in general, the OS will throw away any draw requests it receives above that rate. While there are a select number of people still using CRTs and TV manufacturers are pushing 120hz technology, it's a waste of cpu to generally create animations above 50 to 60 fps.*«

The last point made by Christman is based on the fact that human beings start recognizing sequences of pictures as an animation at about 25-30 frames per second[Web04]. This is also the reason why the european standard for television playback, called PAL, is set to 25fps. Thus, it does not make sense do go far beyond this limit since faster sequences cannot be noticed by human beings anyway. The reason why televisions and computer displays refresh at 50-60 Hz (instead of 25-30) is because lower rates will lead to flickering effects on the screen[Web04]. Spectators will perceive such animations as being »smooth« but the flickering can cause headache and even epileptic seize. Thus, it

[6] 1 fps = 1 »frame per second« drawn by a rendering engine
[7] 60 Hz = $\frac{60}{1s}$ = 60 »screen-refreshes per second« done by a television or computer display.

is important to keep the monitor's refresh rate at least at twice the fps rate, generated by the input signal, if no interpolation technique is used. (Interpolation forces a display to first draw all lines of a screen with un-even numbers. Then, the ones with even values are being shown. This technique can lead to less flickering effects at low refresh rates[Web04].)

In order to create a sequence of pictures with 60 images a second, the presented test variants implement a function which rotates all planets once by 1 degree every 17ms, since:

$$60fps = \frac{60frames}{1s} = \frac{60frames}{1000ms} = \frac{\frac{60frames}{60}}{\frac{1000ms}{60}} \approx \frac{1frame}{16{,}67ms} \approx \frac{1frame}{17ms}$$

Important: This technique **should** result in a frame-rate of 60 fps, but in fact it will lead to a sequence of images displayed in 60 pictures per second. This is not the same as the frame-rate produced by the runtime itself, which makes it unavoidable to differentiate between the desired rate of making changes to the screen and the actual refresh rate forced by the RIA platform. This can be observed in a tutorial video by Lee Brimelow[Bri], a Flash evangelist at Adobe Systems, where this behavior is being demonstrated.

Example: Since Flash does not support multi-threading, its rendering engine runs the so called »drawing-phase« (which refreshes the content that is being displayed on the screen) and the »scripting-phase« (which runs ActionScript-code) in an alternating order[8]. Figure 2.10 on page 50 shows how a scripting phase is being called after frame 2 has been drawn. Although frame 3's visible content does not differ from the one of frame 2, the Flash rendering engine redraws the whole screen. Thus, Flash sometimes reloads frames even if it would not be necessary since no changes to the animations occurred. Hence it is possible that the actual frame-rate can go above the desired 60 fps in the following tests even if the sequence of images was set to the fixed rate of 60 pictures per second.
(**Note:** This anomaly can also occur on other runtimes like e.g. JavaFX)

Compared to WebGL or Java3D, RIA technologies like **Flash** currently don't provide hardware-accelerated 3D-support yet. Although there are some early attempts by Adobe to provide such a framework, the current best-practice for Flash is to use the external library Papervision3D, which emulates 3D-support in software instead of using special GPU functions. This might sound a bit »unfair« compared to hardware-accelerated solutions, but still things could be interesting to compare. The test strategy for Flash was basically the same as on all other RIA runtimes. Due to the usage of other 3D-libraries, the implementation was a bit different. Papervision3D draws its content to a `<mx:Canvas />` component, which seems to be basically the same way FXCanvas3D

[8] This behavior is being described by Thibault Imbert, an employee at Adobe Systems Inc., in his publication »Optimizing performance for the Flash platform«[Imb10]

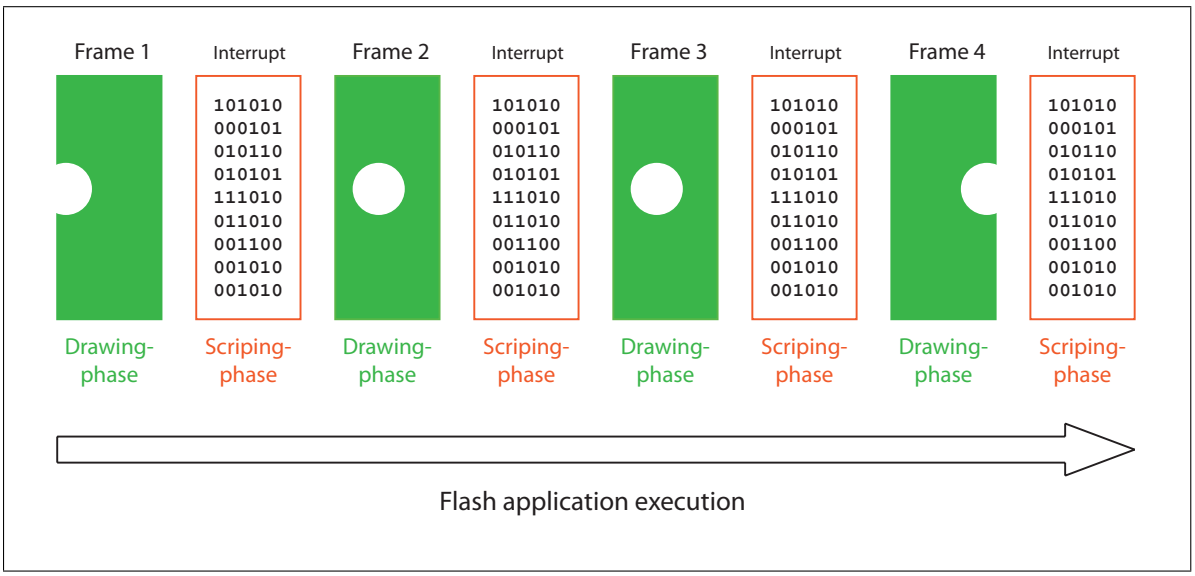

Figure 2.10: Alternating scripting- and drawing-phases of the Flash execution engine.

does it, just without hardware-acceleration. On application start, the whole test is being initialized with 40 planets of size 60, which are covered by their textures using the `BitmapMaterial` object as shown in listing 5.3 on page 127. In order to be able to rotate the planets later, each is put into a `DisplayObject3D`-object (called »pivot«) and then pushed into a global array, which holds all planets in this solar system. Regarding the desired framerate, a timer-event was used which fires every 17ms and leads to a rotation of all planets by 1 degree, which should result in a frame-rate of about 17 fps:

```
// Rotate each planet once every 17ms => 60 fps
var timer:Timer = new Timer(17);
timer.addEventListener(TimerEvent.TIMER, rotate);
timer.start();

public function rotate(event:TimerEvent = null):void {
 for each (var pivot:DisplayObject3D in pivots) {
  pivot.rotationY += 1;
 }
}
```

Also, in order to do the fps measurement, on each `enterFrame` event, the number of frames drawn is increased by 1. After one second, the variable `frames` holds the current fps and can be displayed to the user (see next page for the source-code).

```
// Start getting the current fps
var timer:Timer = new Timer(1000);
timer.addEventListener(TimerEvent.TIMER, getFps);
timer.start();

private function getFps(event:TimerEvent):void{
 frameRateLabel.text = frames + " fps";
 frames = 0;
}
```

Regarding the technical implementation, all RIA runtimes required some kind of external code. This is astonishing since especially **JavaFX** should be able to use the Java3D API with hardware acceleration of the Java Virtual Machine. Unfortunately, JavaFX as well as Java Applets are so called »light-weight applications« which don't support any heavy-weight components for fast 3D rendering. Thus, it was necessary to use an interface called FXCanvas3D[Lam], developed by August Lammersdorf. He bypassed this issue by rendering all the 3D-data offscreen and saving the result to a puffer instead of directly drawing it to the screen. This pixel-information is now being displayed as a sequence of images inside a light-weight JPanel component. Since this object is available inside JavaFX applications, it is possible to integrate this JPanel into a JavaFX scene using a SwingComponent node. FXCanvas3D provides this kind of JPanel and takes care about the synchronization of the render-cycles between Java3D and JavaFX. Lammersdorf also mentioned in an e-mail that this pixel-transport between GPU and CPU does definitely influence the fps. Thus, it was interesting to see if JavaFX would be able to compete against the other RIA solutions even with this handicap. Regarding the rotation operation in Java, all objects had to be grouped together into a `TransferGroup` object and then moved by one `setTransform(Transform3D transformer)` call, as shown in listing 5.1 on page 125. This technique is different than implementations in ActionScript (Flash) or C# (Silverlight), where each planet has to be rotated one by one. This fact has to be kept in mind if it is about to interpret the results of the test since it is possible that the method `setTransform(Transform3D transformer)` shown below has some built-in optimizations which are not being used in the manual rotation techniques in Flash or Silverlight. In order to rotate all planets at once (at desired 60 fps), the following code is being executed every 17 ms:

```
// angle is a global variable of type int
if (angle > 360) angle = 1;
else angle = angle + 1;

// sunTransformer is a global variable of type Transform3D
if (sunTransformer != null) sunTransformer.rotY(Math.toRadians(angle));

// Apply the rotating transformation to the group of planets
if (sunTransformGroup != null && sunTransformer != null)
    sunTransformGroup.setTransform(sunTransformer);
```

A serious issue in order to do the fps measurement was that Java3D did not a offer an event which fires on screen refreshes. Thus, a different technique than compared to Flash had to be used: In order to display 3D graphics to the screen, the mentioned FXCanvas3D component is being added to a object of type `View` by calling `view.addCanvas3D(offCanvas3D)`. This class offers a method `view.getFrameNumber()`, which returns the unique number of a frame in the order they appear on the screen. Saving this value, the number of frames drawn could be counted by checking if the frame-number of the current frame changed. If yes, the number of frames is being accumulated. After one second passed, the number of generated frames is written into a JavaFX component called Label and added to the stage, which is being refreshed every 1000ms. The following listing shows a simplified version of this algorithm:

```
long currentFrameNumber = view.getFrameNumber();
if (currentFrameNumber > lastFrameNumber){
 // A new frame was drawn
 numberOfFrames += currentFrameNumber - lastFrameNumber;
 lastFrameNumber = currentFrameNumber;

 // Update the fps label after 1 second
 if (msElapsed >= 1000){
  fpsLabel.set$text(numberOfFrames + " fps");
  msElapsed = 0;
  numberOfFrames = 0;
 }
}
```

Note: It must be said that it was impossible to run JavaFX applications using the 3D-libraries from interactivemesh.com[Lam] on a Mac. According to the author (August Lammersdorf), this issue is pretty common among Macs since Apple Inc., the manufacturer of Mac OS, does not ship its operating system with Sun's default runtime-environment. Instead, they modify it and release this updated version for their customers in order to better fit the needs of Macintosh computers. Unfortunately, these modifications lead to problems with Java3D in combination with JavaFX. According to Lammersdorf, there is at least one existing workaround he described, like the replacement of Apple's version of Java3D with Sun's original, but this attempt failed on the computer used in this thesis. Thus, no results for the JavaFX 3D-test on Mac OS X can be provided here.

Regarding the **Silverlight** implementation, a framework called Kit3D was necessary, which is basically a ported version of the 3D-library from WPF[9] and thus related to many kinds of C#-based development environments since most of these belong to the family of the .NET framework, which also includes WPF and Silverlight. Kit3D is not a product of Microsoft though. It is a project by Mark Dawson[Daw] who ported the WPF-version of the 3D library `System.Windows.Media.Media3D` to Silverlight. Unfortunately, this framework does not offer much built-in functionality. It is a pure software-emulated 3D-

[9]Windows Presentation Framework

engine ($\Longrightarrow$ no hardware-acceleration) which enables the programmer to draw vector graphic objects (with or without textures) to the screen. The development process is not as easy as using Papervision3D to create a sphere by simply calling `new Sphere()`. Instead, these kind of objects need to be completely created from scratch. Fortunately, Dawson already supplies an example code snippet for creating globes in C# using Kit3D, which was adapted in this work in order to create the planets for the performance test application (The algorithm is based on the book »Charles Petzold's 3D Programming for Windows«). Thus, all that was left to do is to create the solar system by positioning the planets with the help of the already mentioned (pseudo-)random key generator and to implement the rotating algorithm as well as the frame rate measurement. Since the implementation itself is basically the same as already described for Flash (except for syntax differences), a detailed explanation is not given at this point.

Lastly, it must be said that no JavaScript-version of the 3D-test was developed because writing such a benchmark would require a 3D-library like WebGL, which is currently only supported in nightly builds of most browsers. These versions are highly experimental, probably unstable and thus unreliable. Hence it was decided to omit the 3D-test for JavaScript in favor of being able to only use stable versions of currently popular web-browsers.

Results

All 3D-tests provide three values while the benchmark runs, which are the *current* and *average* fps rate, as well as the *time elapsed* (in seconds) since the test was started (see figure 2.8 on page 46). While the current fps value is not vital for the results of this test, the average rate as well as the time-counter are required. The reason for this is that the whole benchmark does not terminate after a fixed amount of time. Instead, it runs indefinitely. Thus, it is important to define a point in time, when to pick which value. For the purpose of this test it was decided to use the average fps rate value after 20 seconds since the start of the benchmark because, it could be observed that usually no more fluctuations occur in most cases after 20 seconds. Even if such changes to the average value occured, they never had a significant impact (more than 1 fps) on the end result.

Having a look at the **final results**, shown in the table in section **??** on page **??**, it could be observed that the Silverlight version of the test performed by far the worst (1-2 fps), no matter on which operating system or web-browser. On the other side, JavaFX is the winner on Microsoft Windows Vista (97 fps on Internet Explorer 8) and Linux (31 fps on Google Chrome 5.0 and Mozilla Firefox 3.6). As already mentioned above, no measurement results on Mac OS X could be achieved. Besides these results, one particular interesting anomaly could be observed running the test on Google's Chrome browser on Microsoft Windows. The result was a frame-rate of 53 fps, which is way less

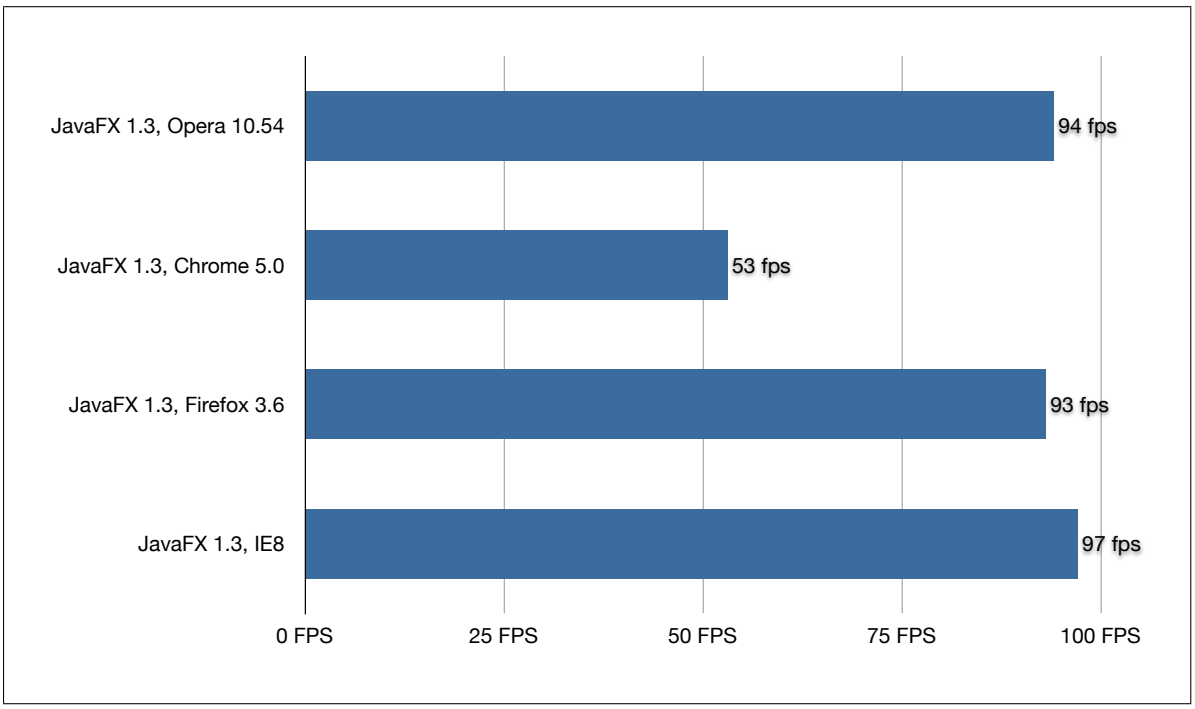

Figure 2.11: Partial results of the 3D-test: JavaFX on Chrome significantly slower than on other browsers.

than compared to the average of the results on other browsers for the same operating system of 94.7 fps (see figure 2.11 on page 54). This test was repeated several times in order to eliminate possible false measurements, but on every run, the result was the same. Based on this insight, it can be said that the web-browser significantly affected the performance on a plugin-based RIA platform in this case.

Examining the results of the Flash platform, it can be said that it often was in average of all RIA technologies. Version 10.0 resulted in an average value of 16.8 fps on Windows while its successor, version 10.1, was slightly faster at an average of 19.3 fps. The behavior on Ubuntu 10.04 was similar (Average of 12,6 fps for both, version 10.0 and 10.1). Having a look at the results from Mac OS X, it must be said that Adobe's new version of its platform had a huge impact on its performance: Version 10.0 ran very slowly at an average rate of 7.6 fps while 10.1 catches up to its Windows- and Linux pendants with an average of 17.8 fps (see figure 2.12 on page 55).

The last obtrusive result in this test was that Flash 10.1 on Opera ran significantly slower (14 fps) than compared to the benchmark in other browsers (average of 18.75 fps). On the one hand side, the difference of almost 5 fps could be explained through browser interferences between its rendering-engine and the Flash plugin-in. On the other hand, Opera 10.54 performed quite normally. This leads to the conclusion that either Opera 10.10 influenced Flash's 3D-capabilities during the test (while Opera 10.54 didn't) or the measurement result must contain false values. Currently, there is a possibility that the

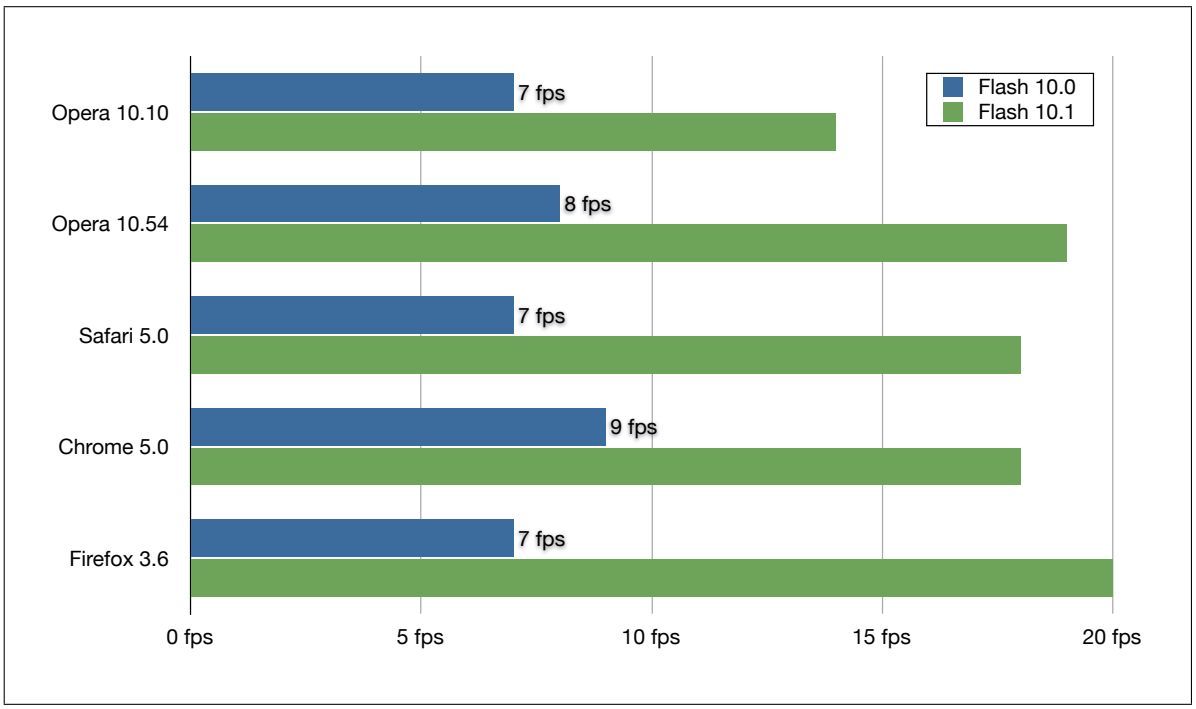

Figure 2.12: Partial results of the 3D-test: Flash Player 10.0 vs 10.1 on Mac OS X 10.6

last point applies because it could be observed that the 3D-animation in Opera 10.54 does subjectively not run as smooth as in other browsers although the benchmark shows an average frame-rate of 19 fps. Apparently, Opera's rendering engine does additional screen-refreshes although they would not be necessary. This anomaly could also be observed in other JavaScript-based benchmarks: When other browsers freeze, Opera often seems to interrupt running algorithms, refresh the screen and then continue processing. It is assumed that this behavior also leads to false fps values in this benchmark. However, since this assumption was not examined any further, the results for both, Opera 10.10 and 10.54 for the Flash 3D-test must be treated with care.

(Note regarding charts in this thesis: These 3D-tests and the 2D pendants are the only ones which produce fps rates instead of ms (milliseconds). This is important to keep in mind regarding graphs illustrated in this publication since higher fps values are better than lower ones (despite the test results measured in ms where lower numbers are better).)

2.3.2 Non-API tests

As the name already says, the following, so called »Non-API tests« were implemented by using as few API-calls (and -classes) as possible. Third-party code was completely avoided. The reason for this decision is that these tests now focus on the actual run-

time environment itself instead of the efficiency of the implementations in the API-layer provided to the programmer. Furthermore, due to the fact that every line in these benchmarks is self-written, the interpretation of the results of these benchmarks should become easier because the source-code is more transparent.

2.3.2.1 Primenumbertest and -generation

Prime numbers play an important role in cryptographic context (like asynchronous methods for key exchange or random key generation, for example), as shown in section 2.3.2.2 on page 59 where two prime numbers p and q are required in order to generate private and public keys for an asymmetric en-/decryption procdure. In the following section, a simple algorithm for generating prime numbers is being implemented, which also includes a test to verify if a given number $i \in N$ is prime.

Test setup

The implementation of the algorithm is basically the same for all RIA solutions except for minimal differences in syntax. Thus, only the source fragments of the JavaScript version will be shown here. Regarding the JavaFX version of the test, it is important to mention that this whole test was written in JavaFX Script. No »plain Java« was used here. Although it is not possible (yet) to efficiently test if a value $n \in \mathbb{N}$ is prime, the basic attempt is rather simple: The naive test, for verifying if a number $n \in \mathbb{N}$ is prime, is based on the verification if there is no element $i \in \mathbb{N}_0$ which fulfills $n \; mod \; i = 0$, except for 1 and n itself.

(**Note:** At this point, it is important to mention that it is clear that this algorithm definitely is not the most efficient. Due to the purpose of this benchmark to be a stressful test for the CPU, this fact is irrelevant though.)

In JavaScript, for example, this can be implemented pretty easily by iterating from $2...(n-1)$ and checking for each iteration if $n \; mod \; i = 0$ applies. This algorithm works similar with all other imperative programming languages, but for the purpose of this demo, only the JavaScript version is shown in the listing below:

```
function isPrime(n){
 for (var i=2; i<n; i++)
  if ((n % i) == 0){
   return false;
  }
 }
 return true;
}
```

Using the above algorithm, prime numbers can be generated by iterating over a variable m and checking for each loop if m is prime or not. This must be repeated until enough numbers were generated:

```
var result = findPrimes(200); // Find 200 primes

// Returns an array with n prime numbers
function findPrimes(n){
 var primes = array();
 var index = 0;
 for (var i=2; i<n+2; i++){ // 0 and 1 are not primes => Skip them
  if (isPrime(i)){
   // Store found primes to the array
   primes[index] = i;
   index++;
  }
  else n++;
 }
 return primes;
}
```

Results

On **Mac OS X 10.6**, it can be said that the best results come from all JavaScript-versions except for Opera 10.10 (2487ms) while version 10.54 takes the lead at only 97ms. Regarding the results on Silverlight, it must be said that it is astonishing that all tests ended with exactly 192ms. Currently, it is unknown if there were any caching activities during the tests although caching was disabled. However, regarding the performance, it can be said that Silverlight finished the prime generation tests very quickly and belongs to the winners of this test. The most astonishing result on Mac OS X though was that JavaFX took only 288ms on Apple's Safari browser while the plug-ins for Opera 10.54 and Firefox 3.6 needed 1787ms and 1454ms. Unless there were no measurement errors, this can be considered as proof that browsers can influence the performance of plugin-based applications running inside. For JavaFX on Chrome 5.0 as well Opera 10.10 no results could be measured due to the lack of plug-in support. The same applies for Silverlight on Chrome 5.0. Regarding the results on Flash, it could be observed that version 10.1 did a huge boost on Mac OS X (at an average of 441ms) compared to Flash 10.0 (at an average of 3306ms).

On **Microsoft Windows Vista**, the results were a little bit different: Worst numbers come from the JavaScript-versions of Internet Explorer 8.0 at 4921ms as well as Opera 10.10 at 3103ms. In contrast, all other browsers performed well at results between 207ms and 399ms. Together with Silverlight (204ms - 248ms) and JavaFX (186ms - 359ms) these technologies take the lead on Windows. Mostly not expected was the fact that Flash 10.0 (average of 789ms) performed a little better than its successor 10.1 at 830ms. The difference is not huge but currently there is no explanation why the new version is not

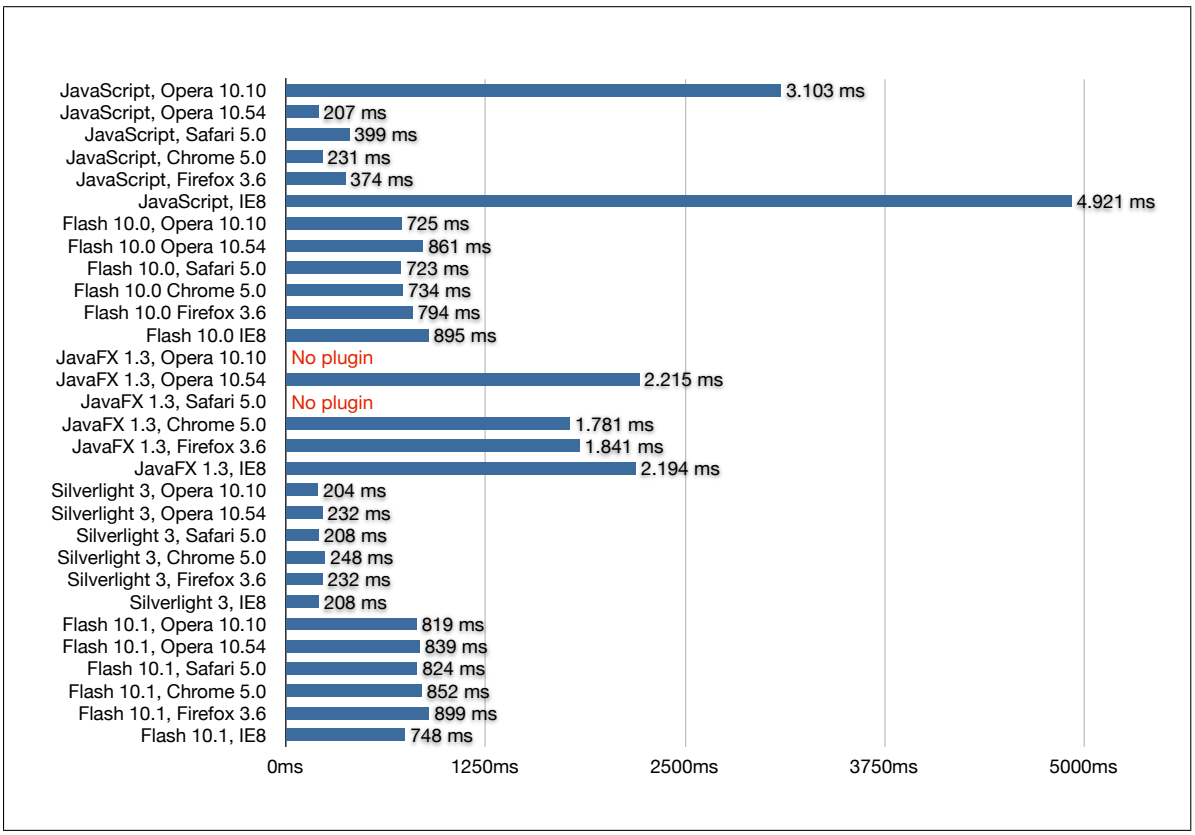

Figure 2.13: Results of the prime-test on Microsoft Windows Vista

faster but actually even slower than its predecessor (see figure 2.13 on page 58). Looking at the results of JavaFX, it must be said, that these are very unusual. Normally, all results across various operating systems are similar but in this case, JavaFX performed as one of the worst platforms, right behind the JavaScript-versions on Opera 10.10 and Internet Explorer 8.0 while the Mac- and Linux-pendants did way better.

Having a look at the results on **Ubuntu 10.04**, it can be said that Moonlight, the Mono-version of Silverlight for Linux, again performed best. Unfortunately, there are only results for this test on Firefox 3.6 since there is no Moonlight plug-in for other browsers currently available. Similar to the results from other operating systems, Opera 10.10 again returned the worst result at 4114ms on JavaScript while Google Chrome 5.0 at 165ms and Firefox 3.6 at 279ms are at the top of the field. As expected, Flash 10.1 (average of 552 ms) performs about two times better than 10.0 (average of 1028 ms)

2.3.2.2 Prime factorization

One purpose of prime factorization for example is the calculation of the greatest common divisor (GCD), which plays a role in some cryptographic functions, like RSA[10], an asymmetric key exchange algorithm for generating public and secret keys in order to be able to en-/decrypt data, as well as digitally sign documents. The algorithm works as described in the following paragraph[Kar07]:

1. Choose two prime numbers $p, q \in \mathbb{N}$
2. Get $n = pq$
3. Get $\varphi(n) = (p-1)(q-1)$
4. Choose e so that $gcd(e, \varphi(n)) = 1$
5. Get e so that $ed \equiv 1 \ mod \ \varphi(n)$ by using the extended euclidian algorithm
6. $C = M^e \ mod \ n$ and $M = C^d \ mod \ n$
7. Generate the public key $P = (e, n)$ and the secret key $S = (d, n)$

Prime factorization is not only needed for generation of these key pairs. It also plays an important role regarding the security of RSA. As seen in step 2, the calculation of n is being done through $n = pq$. In order to break the algorithm, all steps must be re-done backwards. Since both, p and q are prime numbers, their calculation requires the prime factorization of n, which has exponential runtime complexity and can thus not be calculated efficiently.

Now, talking about RIAs, these cryptographic problems might not immediately come into one's mind, but there are some cases where this can be useful, like for example:

- **Example 1:** E-Mail client

  Imagine a web-mail client based on a random RIA technology. Let's say this applications supports PGP[11]. Then, the generation of key-pairs (for example using RSA) would be a nice-to-have feature.

[10] Rivest, Shamir and Adleman

[11] Pretty Good Privacy, an asymmetric en-/decryption as well as signature verification technology. See *http://www.ietf.org/rfc/rfc4880.txt*

- **Example 2:** SSL/TLS

  Imagine a random application trying to connect to a host using SSL/TLS, without using the implementation of third party software (e.g. a browser where the RIA runs in). The SSL/TLS technology works basically as follows[Kar07]: First, a symmetric key is being exchanged between two communication partners by using an asymmetric exchanging procedure. This is also called the »handshaking process«. Then, due to performance benefits of symmetric en-/decryption technologies compared to asymmetric ones, a cryptographic algorithm like AES[12] is used in order to secure the actual data. Since one of the possible handshaking algorithms is RSA, the above mentioned issues regarding gcd-calculation apply.

Test setup

The implementation of the prime factorization test is based on the *trivial division method*. This is definitely not the fastest algorithm but it works well in order to get small factors, which is just right for these experiments. Again, the purpose of these test implementations is to see how long specific RIA solutions need to compute various algorithms and not to use the fastest algorithm available. The trivial division method works as follows:

Be p a prime number and M a sequence defined through $\mathbb{N} \supset M = \{2, 3, 4...(p-1)\}$. If there are at least two $n, m \in M$ with $p = nm$, then p can be factorized. Regarding the implementation, the easiest way doing this is to iterate over a variable `i`, starting at 2, and check each time if $p \; mod \; i = 0$. If yes and $j = \frac{p}{i}$ is also a prime number, p can be factorized into `i` and `j`. Now, the procedure must be redone with p replaced by j. If no further prime factor was found, the algorithm continues with the next prime number which fulfills $p \; mod \; i = 0$ (for the original p). The algorithm terminates if $i > \sqrt{p}$. The remaining number is then a prime number and the last prime factor of p at the same time. The JavaScript source of the implementation can be examined in listing 5.4 on page 127. Other versions in Silverlight, JavaFX and Flash look very similar and are thus omitted in this section.

Results

On **Mac OS X 10.6**, Opera 10.10 on JavaScript again was the slowest browser at 1123ms while others performed very well between 47ms (Opera 10.54) and 133ms (Firefox 3.6). Flash 10.0 was the slowest solution by far at an average of 1589ms while Flash Player 10.1 catches up to its competitors with an average of 217ms. This results in a performance boost of factor 7.3 from version 10.0 to 10.1 (see figure 2.14 on page 61). The winners of

[12] Advanced Encryption Standard, the successor of DES, a symmetric en- and decryption algorithm

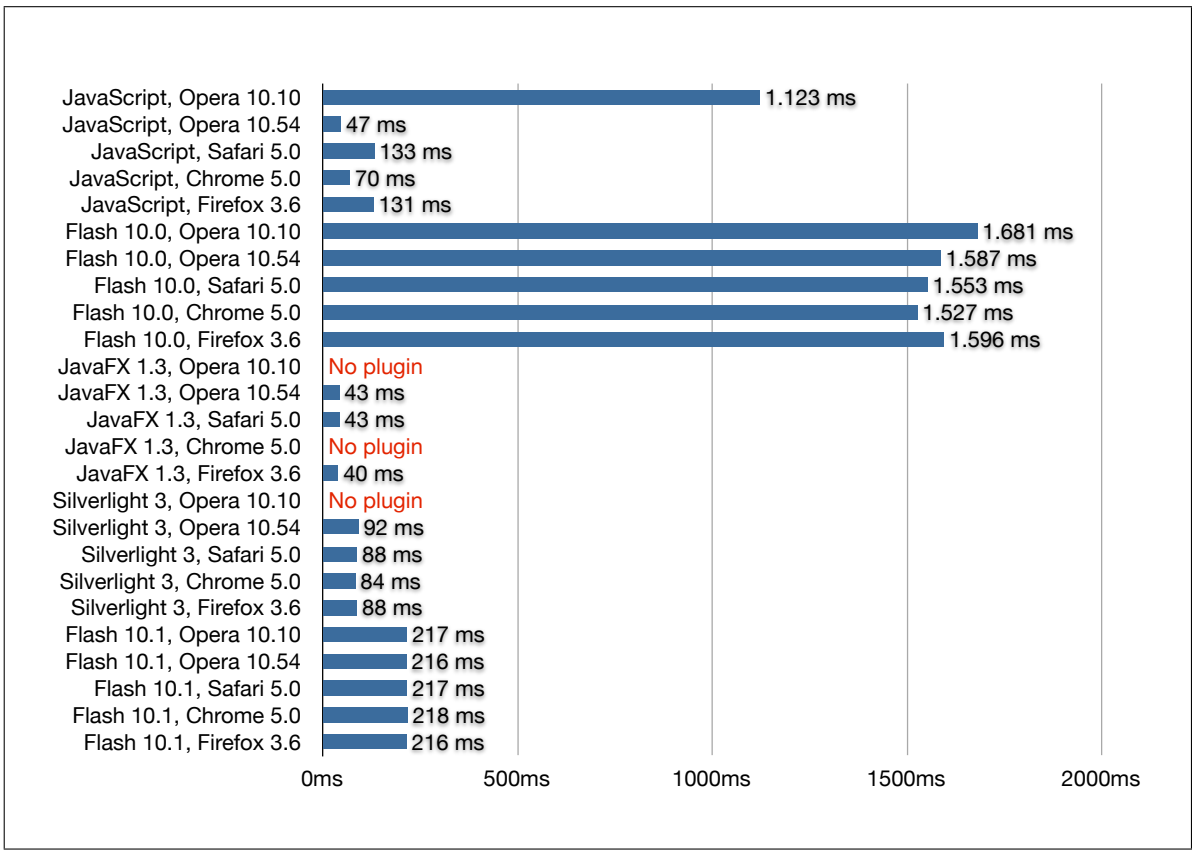

Figure 2.14: Results of the prime-factorization-test on Mac OS X 10.6

this benchmark on Mac OS are JavaFX at results between 40ms - 43ms on Opera 10.54, Safari 5.0 and Firefox 3.6, followed by Silverlight with numbers between 84ms and 92ms.

Having a look at the results on **Microsoft Windows Vista**, Opera 10.10 and Internet Explorer 8.0 performed worst on the JavaScript-test at 1411ms and 2290ms, as expected. All other browsers did quite well, especially Mozilla's Firefox 3.6 which was even 55ms faster than Apple's Safari browser. Obviously, mathematical calculations fit this browser well. Regarding, JavaFX and Silverlight, it can be said that both platforms again take the lead at average values of 59ms (JavaFX) and 146ms (Silverlight). Regarding Flash performance, it was noticed that there was no significant difference visible between version 10.0 (average of 383ms) and 10.1 (average of 376ms). Obviously, Adobe did not put much effort regarding enhancements related to this test for the Windows version of their new Flash Player. Other obtrusive observations could not be made on this operating system.

The results on **Ubuntu 10.04** do not much differ from previously made observations on Mac OS and Windows. Again, Opera 10.10 performs badly on JavaScript while both, JavaFX as well as Moonlight, return the best values, no matter on which browser.

Regarding Flash, it can be said that the new version 10.1 seems to have a way bigger impact on Linux than on Windows at an average rate of 260ms while 10.0 was actually 3.3 times slower.

2.3.2.3 (Pseudo) Random key generation

Random key generators are often used in a cryptographic context, for example during handshaking processes, which require random numbers in order to verify the so called freshness of keys. But this is not the only use-case. As seen in section 2.3.1.3 on page 43, placing planets in the 3D-experiment also relied on this generator.

The algorithm used in this thesis is called the *linear congruency method* and produces a sequence of (pseudo-)random numbers based on an initial value, called the seed. The term »pseudo« means that the sequence of numbers are not really random, since they are re-produceable if the same seed is given. »True« randomness is hard to create using computers and thus often relies on already random input (e.g. entropies over keyboard typing patterns of the user). Since all tests must be comparable, the fact that the same sequence of random values re-occurs using the same seed, is just right. The linear congruency method, as described in »Kryptographische Zufallszahlengeneratoren« by Ralf Jungblut[Jun95], is very easy and efficient and thus often implemented in common software projects. Based on the seed value $y_0 \in \mathbb{N}_0$, the sequence of random numbers is being recursively generated by calculating:

$y_k = (ay_{k-1} + b)\ mod\ m$ with $1 < k \in \mathbb{N}$

For each iteration, $y_k \in \{0...m\} \subseteq \mathbb{N}$ applies.

a, b and m are constants which must fulfill the following conditions:

- $m > 0$ and »very large« (See below for an example)
- $0 \leq b < m$
- $1 \leq a < m$
- $gcd(a, m) = 1$

(gcd = »Greatest Common Divisor«)

Test setup

The implementation in this thesis is based on the following values for the constants a, m and b (as recommended by Jungblut).

- $a = 513$
- $m = 2^{48}$
- $b = 29741096258473$

(**Note:** Since the implementation of the algorithm is very similar on all RIA solutions, only the (simplified) **JavaScript** version is shown here (see listing 5.6 on page 128).)

Both, the Flash implementation as well as the JavaScript version output all generated values after the algorithm terminated. This feature had to be removed though. The section »Results« below gives a detailed explanation why this was done.

Results

First of all it must be said that early versions of this benchmark dumped all random numbers to the screen so that the user can see that the generation was successful. In newer versions of the test, this feature was omitted because it took most RIA runtimes very long to output all generated values. Based on this insight, it was assumed that dumping a lot of strings one-by-one to a text-component, like a TextArea for example, required a lot of CPU power. Thus, the so called »StringGUIPushTest« for the series of Focus-tests, as described in section 2.4.1.1 on page 86, was developed.

Having a look at the results of this random key generator test, the following observations could be made on **Mac OS X 10.6**: Again, Opera 10.10 on JavaScript performed worst at 2047 ms. All other browsers did quite well at rates between 106ms (=Minimum value, Safari 5.0) and 315ms (=Peak value, Google Chrome 5.0). It's is also interesting to see that Firefox 3.6 did quite well at 119ms, as it could already be observed in the prime-tests in the prior sections. The second worst results come from Adobe's Flash Player 10.0 at an average value of 819ms over all browsers. It is astonishing to see how its successor experiences a huge performance boost by factor 4.6. Regarding JavaFX and Silverlight, both technologies again perform best at average rates of 89ms (JavaFX) and 98ms (Silverlight). The only downside is again the lack of plug-ins for some browsers like Opera 10.10 and Google Chrome 5.0.

On Microsoft's **Windows Vista**, similar results could be observed: Opera 10.10 and Internet Explorer 8.0 performed worst at 1317ms and 2192ms while JavaFX and Silverlight

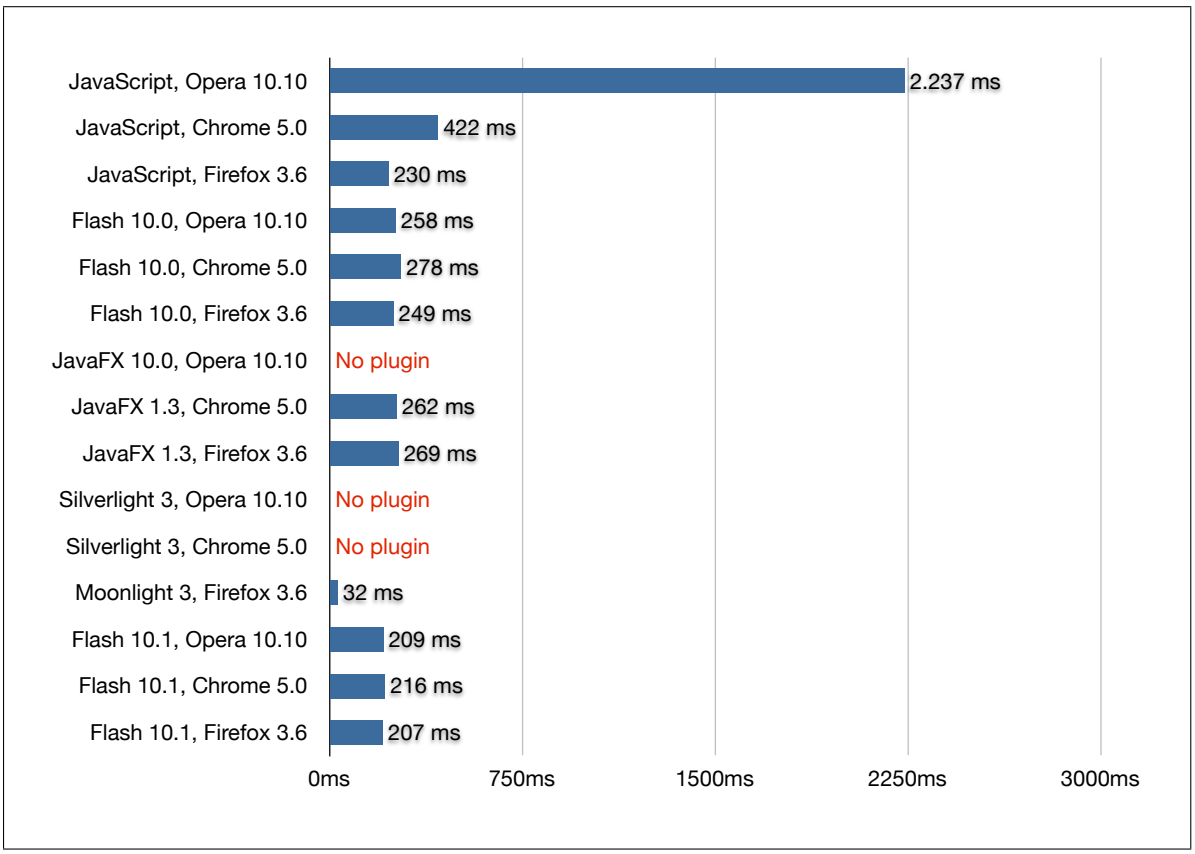

Figure 2.15: Results of the random-key-generation test on Ubuntu Linux 10.04

again took the lead, no matter on which browser. Flash Player 10.0 was in average of all results while version 10.1 was about 20-30% faster. No further obtrusive observations were made on this operating system.

On **Ubuntu Linux 10.04**, a different picture than compared to Windows and Mac OS could be observed. Besides the fact that Opera 10.10 was extremely slow again, all other technologies were almost even (see figure 2.15 on page 64), except for Moonlight on Firefox 3.6 at very fast 35ms and Google Chrome 5.0 as the slowest browser in the field of JavaScript-tests for random key generation.

2.3.2.4 Run length encoding

Run length encoding, also simply called RLE, is a classic, lossless compression algorithm based on the idea of merging together recurring sequences in data streams by removing these (except for one representative) and adding the number of re-appearances to it[Web04].

Example:

$b\underbrace{aaaaa}_{=5a}fdhgfijfjdkletigidf\underbrace{kkk}_{=3k}ddkdkkee\underbrace{www}_{=3w}nndfjgjkgllsaa$ (Length = 53)

This string can be compressed to:

$b5afdhgfijfjdkletigidf3kddkdkkee3wnndfjgjkgllsaa$ (Length = 49)

$\Longrightarrow$ Compression by 7,5%

The decompression process is trivial. By simply expanding the packed parts, based on the given number for each sequence, the original stream can be retrieved. This algorithm itself is effective if there are many redundancies in the given data stream, like white spaces in program source code, for instance. Without these recurrences, RLE does not compress well. The reason why this experiment was added to the series of performance tests is because RLE is a good example for working with data streams in strings. In order to find redundancies, the data stream must be examined for any recurrences. For the actual compression, the replacement of characters must be implemented. Depending on the type of input stream, these operations can become quite nasty: While linear lists offer easy and effective element replacement functions, strings and arrays are more problematical. For the purpose of this experiment, it was decided to use strings containing bit-streams in order to verify how each single RIA platform can handle these kind of big variables.

Note: It is clear that implementing RLE using other data structures like linear lists would be much more efficient since the splitting and concatenation operations would then have constant run time complexity of $O(1)$. The reason for this is that string-operations are usually based on call-by-value while list-operations use call-by-reference. Thus, list-operations only need to copy memory addresses instead of the actual object itself, but since the purpose of this benchmark is to see how the various RIA platforms can handle String operations, RLE was implemented using Strings.

The actual implementation is not as easy as mentioned above, since it is possible that numbers also belong to the original input stream. This issue can cause problems if it is about to determine the length of compressed sequences.

Example:

Given is a bit-stream S = "010$\underbrace{111111111111110}_{(14)1}$", which should be compressed.

Using the RLE algorithm, this string can be compressed to S' = "0101410".

The problem now occurs in the process of decompression: Which of the below is correct?

$$0101\underbrace{41}_{=1111}0$$

or

$$010\underbrace{141}_{=11111111111111}0$$

This happens if numbers are also part of the input stream with sequence lengths greater than 9 because then it cannot be determined if the value, which represents the number of recurrences, has one or more digits. In mathematical terms: The mapping between original and compressed version is not surjective. Thus, it is important to choose two characters, which do not appear in the stream, in order to mark the start- and end-point of a compressed sequence. This leads to weaker compression rates since only sequences with a length of at least 4 are worth to be encoded. In this experiment the used markers are "A" (for sequences of 1) and "B" (for sequences of 0) since these characters do not appear in streams consisting of bits.

Example:

The previously mentioned stream S can now be compressed to S' = "010A14A0" and uniquely decoded back to its original representation since it is clear that the value between the two A's must be the desired sequence length.

Test setup

Regarding the implementation itself, the input data stream was created by reading in a ASCii test file of 12 kb size and converting it to its bit-stream representation. The result was then stored into a variable of type String. The basic idea of the implementation is as follows:

1. Iterate over each character inside the string and let a function check for redundancies in this area (See line 3 in the source below).

2. If redundancies were found, call a function (with the index position as parameter), which does the actual compression (See line 8 in the source below).

3. When the compression is done, go to the next character in the String and continue searching for redundancies.

The JavaScript version of the implementation is shown in listing 2.1 on page 67. The other versions created in C#, AS3[13] and JavaFX Script work similar.

```
1  for (var i=0; i<bitstream.length; i++){
2   // Check if there are redundancies following after index i
3   var numOfRedundances = getNumOfRedundances(i, bitstream);
4
5   // Only sequences with length > 3 are worth to be compressed
6   if (numOfRedundances > 3){
7    // Do the actual compression
8    bitstream = compress(i, numOfRedundances, bitstream);
9   }
10 }
```

Listing 2.1: Finding re-occurrences of single bits inside a bitstream (JavaScript version)

The function `getNumOfRedundances(i, bitstream)` simply checks if the characters in `bitstream` after position `i` are the same as in `bitstream.charAt(i)`. If yes, the number of recurrences is returned. The method `compress(i, numOfRedundances, bitstream)` now converts these sub-strings into their RLE-encoded representation, as described in the previous section. Since both functions only use some simple String-operations like `concat(str1, str2)`, `substr(int1, int2)`, `indexOf(char)` or `charAt(pos)`, the full source is not shown here, but can be downloaded from *http://www.timo-ernst.net/riabench-start*.

Results

The results of the run-length-encoding test are the first ones which seem to differ a lot to previously made observations. On **Mac OS X**, the worst results come from JavaFX (on all browsers) at rates between 1705ms - 1826 as well as Flash Player 10.1 at an average value of 1756ms. It was astonishing to see, that even the older version 10.0 performed better than its successor at an average of 1378ms. The best results of the field come from the JavaScript-versions at rates between 399ms (Chrome 5.0) and 1071ms (Opera 10.10). For a better overview, these values can be seen in figure 2.16 on page 68.

Looking at the results from **Microsoft Windows Vista**, it can be said that the results here are even more balanced except for the JavaScript-versions which again take the lead (see figure 2.17 on page 69). Even Internet Explorer 8.0 and Opera 10.10 performed better than all other platforms. Obviously, many RIA technologies seems to encounter heavy problems if it comes up to working with strings, since the run-length-encoding benchmark heavily relies on such operations. At this point, it can be interesting to examine the results of the Focus-test for working with strings (see section 2.4.2 on page 90).

[13] ActionScript 3

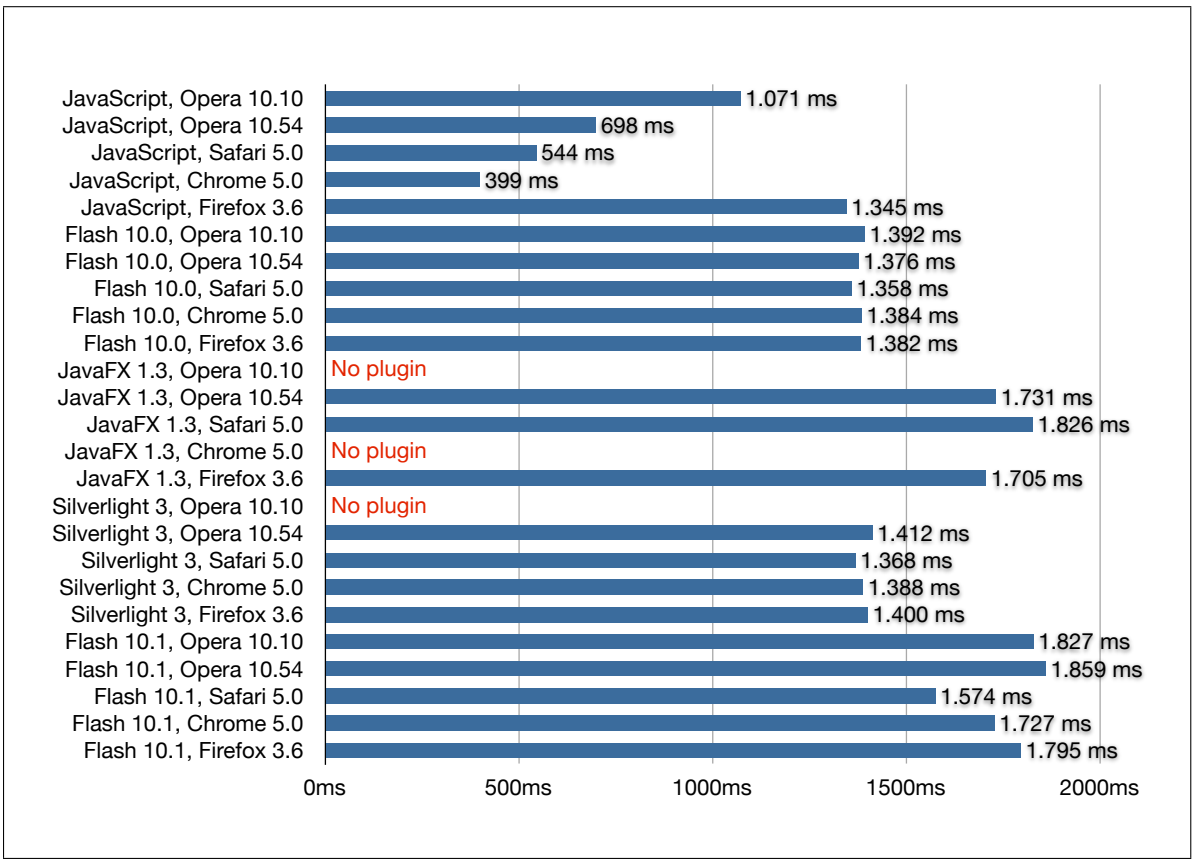

Figure 2.16: Results of the run-length-encoding-test on Mac OS X 10.6

On **Ubuntu 10.04** similar observations like on Windows and Mac OS could be made. Google's Chrome 5.0 browser was by far the fastest solution at 375ms processing time, followed by Opera 10.10 and Firefox 3.6 (both JavaScript) at 951ms and 987ms, as well as Moonlight on Firefox 3.6 at 1032ms. All other results were below this value. Worst results come from Flash 10.1 at an average number of 1236ms, although even its predecessor, version 10.0, was faster at rates between 1227ms - 1242ms.

2.3.2.5 2D acceleration

As described in section 2.3.1.3 on page 43 about the importance of 3D-support in Rich Internet Applications, the ability to draw graphical objects to the screen is absolutely vital for the web (e.g. in browser-games). But not only video-games benefit from fast rendering engines. »Ordinary« applications often rely on transitions like zooming or fading effects in order to increase usability of their user-interface. Now, if to many frame-drops occur during such transitions, the consequence is often that animations begin to stutter, which again leads to the opposite of what was originally intended: Less usability. Usually, the cause for these kind of problems is the lack of appropriate hardware but

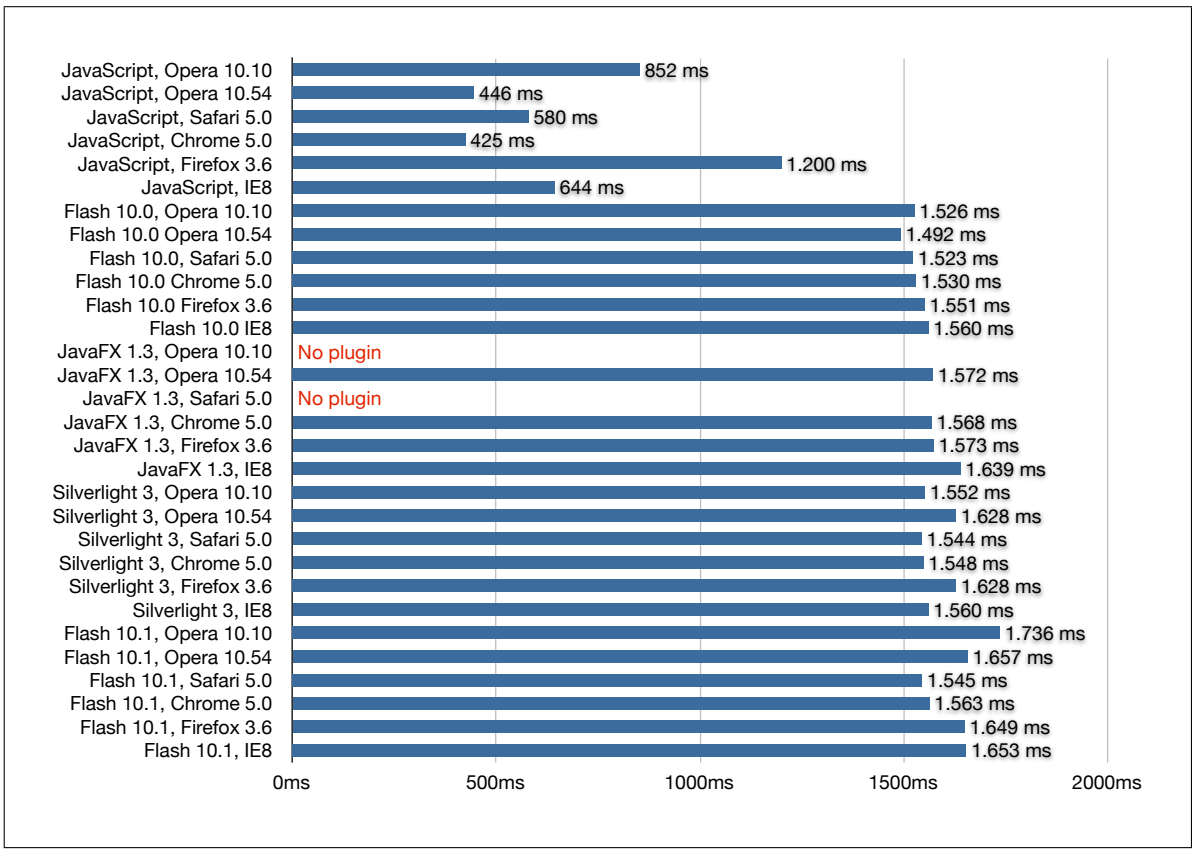

Figure 2.17: Results of the run-length-encoding-test on Microsoft Windows Vista

sometimes the implementation of the drawing engines, which produce the animations on screen, are implemented inefficiently. In order to verify how the RIA-solutions perform regarding 2D-graphic acceleration, a dedicated test, as described below, was developed:

It is clear that the requirements to hard- and software increase, the more objects a 2D-engine has to render. Thus, the 2D-stress-test simply generates a lot of small rectangles, draws them to the screen and then moves these from top to bottom (See 2.18 on page 70 for a screenshot of the JavaScript-version). When all these »particles« reached this destination point, the benchmark has finished. While the test is running, two important measurement-values are being shown on-screen. One is the *current* rate of fps[14] and the other is the *average* fps rate. Although only the last one is really important for the result of the test, the current fps value could be interesting in order to see how and when frame-drops occur, depending on the number of objects being displayed on screen.

[14]Frames Per Second

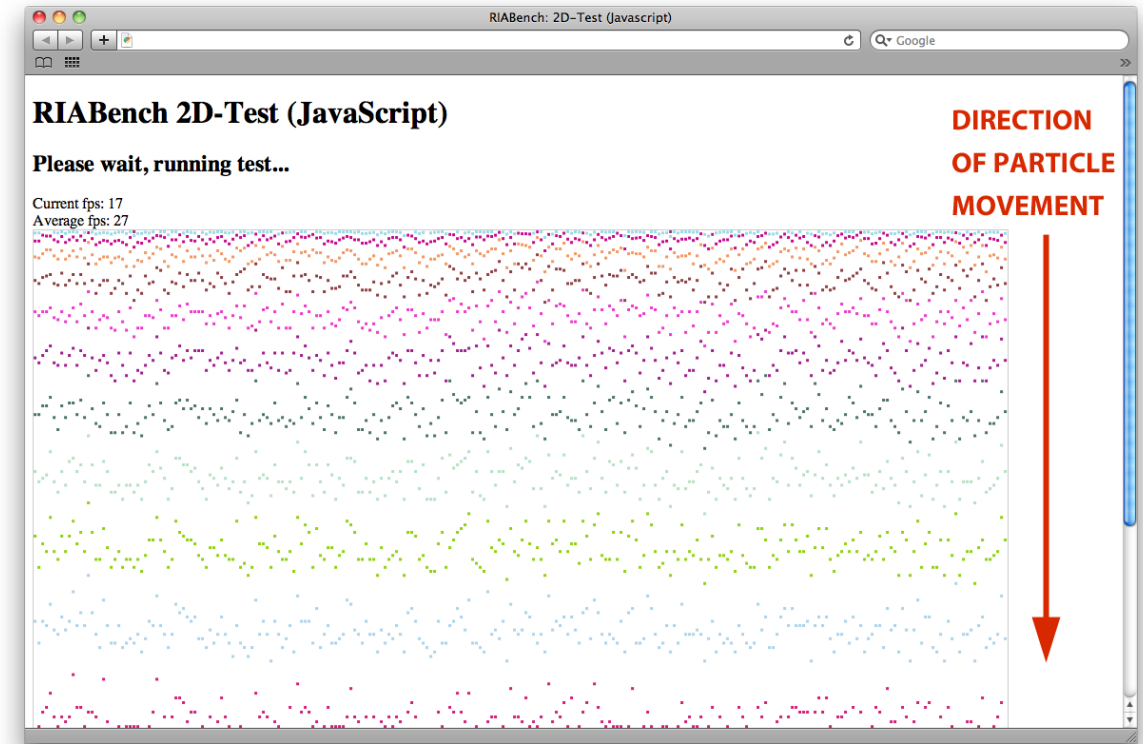

Figure 2.18: Screenshot of the JavaScript-version of the 2D-stresstest

Test setup

The »stage« where the whole animation will play on has the dimension of 1024 pixels in width and 768 in height. All particles (=rectangles of size 3x3px) are initially being positioned horizontally on top of the canvas. Between each of these, a small gap of 1px helps to differentiate each object from its neighbor. This set-up leads to a row of »particles« where each row has its own random particle-color, which means that all particles belonging to one row, have the same color value. The reason why each row has its own color is that it can be handy to be able to visually differentiate between single particles and to be able to observe the way the animation plays. The color value for each row is being randomly generated, thus each row has a new color value for each run at a very high probability. In order to increase the number of objects to move, the number of rows was fixed to 20. Having these values, the number of particles can be calculated as follows:

$$
\begin{aligned}
&stageWidth = 1024 \\
&particleWidth = 3 \\
&gap = 1 \\
&numberOfRows = 20 \\
\Longrightarrow\ &numberOfParticles = 20 * \tfrac{1024}{3+1} = 5120
\end{aligned}
$$

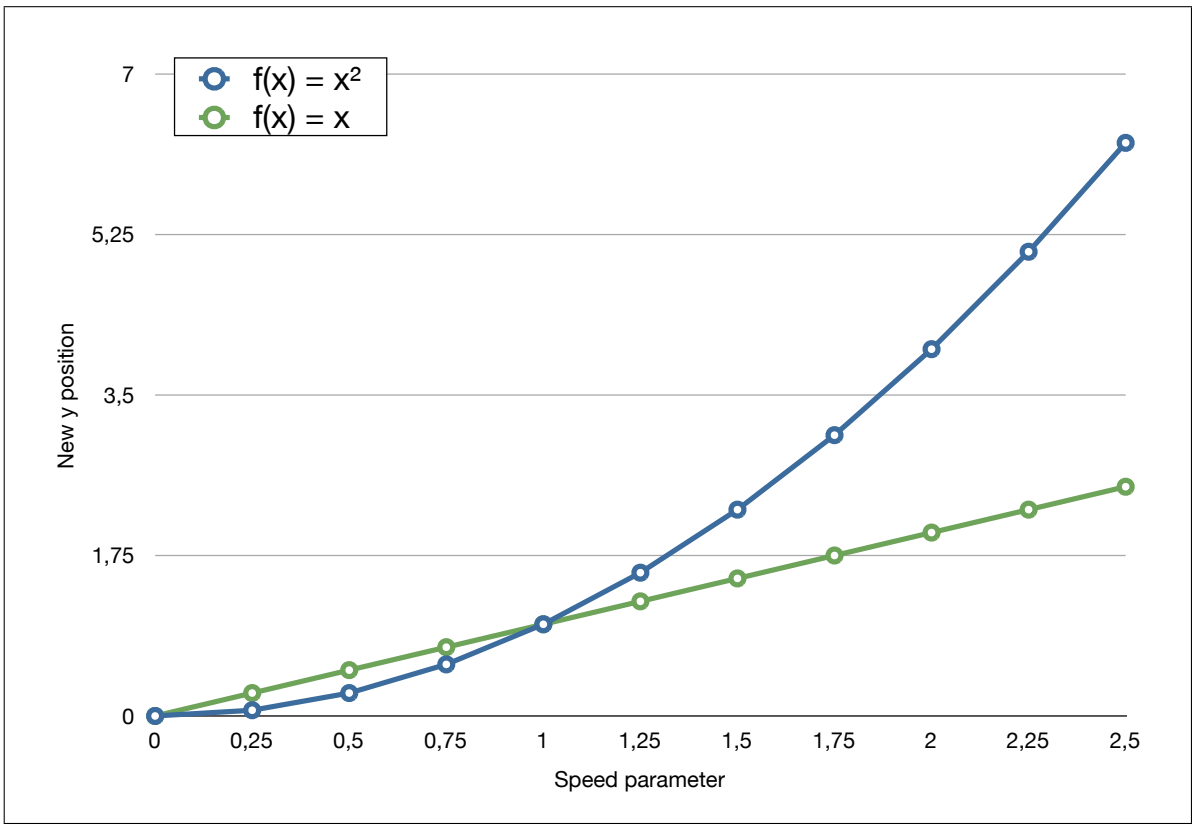

Figure 2.19: Linear vs quadratic transitions

Of course, this value can be easily changed by either manipulating the stage-size, particle-size, gap-size or the number of rows, but in order to get comparable results between the various RIA technologies tested in this thesis, these value were fixed.

Regarding the animation itself, all rows are in order. There is a first, a second and of course last row. Once the benchmark starts, the first particle of the first row will start to move to the bottom at a chance of 3:7. Then it's the second particle in the same row which will move at the same chance but only if its forerunner (=the particle in the same position of the row before) hasn't started yet. Once all particles of one row were either moved or not, the second row will be stepped through the same way and so on for all remaining rows. When the last row is done, the algorithm re-starts from beginning until all particles reached the bottom of the stage. To understand this easier, a short piece of pseudo-code (see listing 5.5 on page 128) illustrates the described algorithm. The reason why each particle starts with a chance of 3:7 is simple: If every rectangle would start moving immediately when it's being called, all that one would see is a pile of rows dropping from top to bottom and nothing more. Since the purpose of this test is to move these particles as a big cluttered cluster, it is necessary to delay the start-time of some particles. This leads to a nice and screen-filling animation. The value »3:7« was achieved by simply testing various rates. Higher rates, like e.g. 6:4 would lead to too short overall

animation time for the benchmark because most of the particles would start to early. Thus, they'd reach the bottom very fast and the stress-test would end to quickly. On the other side, if the rate was to low, like e.g. 1:9, the particles would be apart to far from each other, which lead to a long duration of the whole test. Furthermore, the computer, this benchmark runs on would not be used to its maximum capacity since the number of moving objects would be to low.

As described in section 2.3.1.3 on page 48, the maximum number of screen-refreshes, also known as »fps[15]«, is limited to 60 in order to not exceed the build-in limitations of some RIA-platforms regarding the screen-repetition-rate. Thus, each animation step is being done every 17ms, since at a desired rate of 60 frames per second, the required interval for the screen-refreshes can be calculates as follows:

$$60fps = \frac{60frames}{1s} = \frac{60frames}{1000ms} = \frac{\frac{60frames}{60}}{\frac{1000ms}{60}} \approx \frac{1frame}{16{,}67ms} \approx \frac{1frame}{17ms}$$

Having a fixed screen refresh rate now, the last thing remaining in order to get the particles animated, is to set the speed at which they are supposed to move. Thus, each particle object has a new property called »speed«[16]. This variable is being instantiated with the default value 1. Whenever a particle gets moved now, its speed value gets first incremented by 1 and then quadratically increased. The result is the new current vertical position of the particle. This quadratic acceleration leads to are smoother movement than linear transitions of the object because the animation first starts moving slowly (for $x \leq 1$) but then becomes quickly faster for $x > 1$ (see figure 2.19 on page 71).

Example: Given is a particle-object with the initial property values *speed=0* and *y=0*. Every 17ms the screen gets refreshed as visualized in the table below. — Note that the following example starts with the default value 0 and that each step leads to an incrementation of 0.25 instead of 1, which simply makes figure 2.19 on page 71 easier to read. Of course, the resulting new values for the y-coordinate had to be rounded because there is no such thing like a half pixel.

[15] Frames Per Second

[16] May have a different name in some implementations. For example, in the JavaScript-version this property is being called »timeElapsed« since its value is being incremented every time the internal timer event fires and forces the screen to refresh. Its purpose remains the same as described here though.

Speed	**New y position** (rounded)	**Time elapsed**
0	0	0 ms
0,25	0	17 ms
0,5	0	34 ms
0,75	1	51 ms
1	1	68 ms
1,25	2	103 ms
1,5	2	120 ms
1,75	3	137 ms
2	4	154 ms
2,25	5	171 ms
2,5	6	188 ms
...	...	...
n	Canvas bottom	Termination

The last step of the whole procedure is to destroy the particle object when it reaches the bottom of the stage. It is important to know, that this will not influence the rectangle which was drawn to the canvas, since it is just a visual representative of the particle object. Thus, the rectangle will remain at the bottom and the particle object itself can be destroyed in order to prevent memory leaks through redundant object clutters.

The implementation of the test for **JavaScript** was pretty straight-forward and complies very well to the theoretical idea as described above. Since HTML5 finally includes a canvas-element and most browser manufacturers did already implement this feature, it is easy to use this as the stage in order to draw 2d graphics on it. Rectangles can be painted using the method `fillRect(x, y, width, height)` of the canvas's context property. Removing particles is as easy as drawing them using the `clearRect(x, y, width, height)`-method. Although there is no function provided by the JavaScript API in order to move objects such a function can be easily implemented manually:

```
function moveRect(fromX, fromY, toX, toY, width, height){
 clearRect(fromX, fromY, width, height);
 drawRect(toX, toY, width, height);
}
```

Thus, it is not required to delete the whole stage in order to animate objects on the canvas. Regarding the result of the test, it is expected that the JavaScript-version of this test should benefit from this fact, while other platforms (like Adobe Flex) must refresh the whole stage in order to move objects.

Lastly, particles, which have reached the bottom of the canvas are being deleted using the `delete` operator. The verification if a object can be destroyed is being done every time it is supposed to be moved on stage (see next page for the source-code).

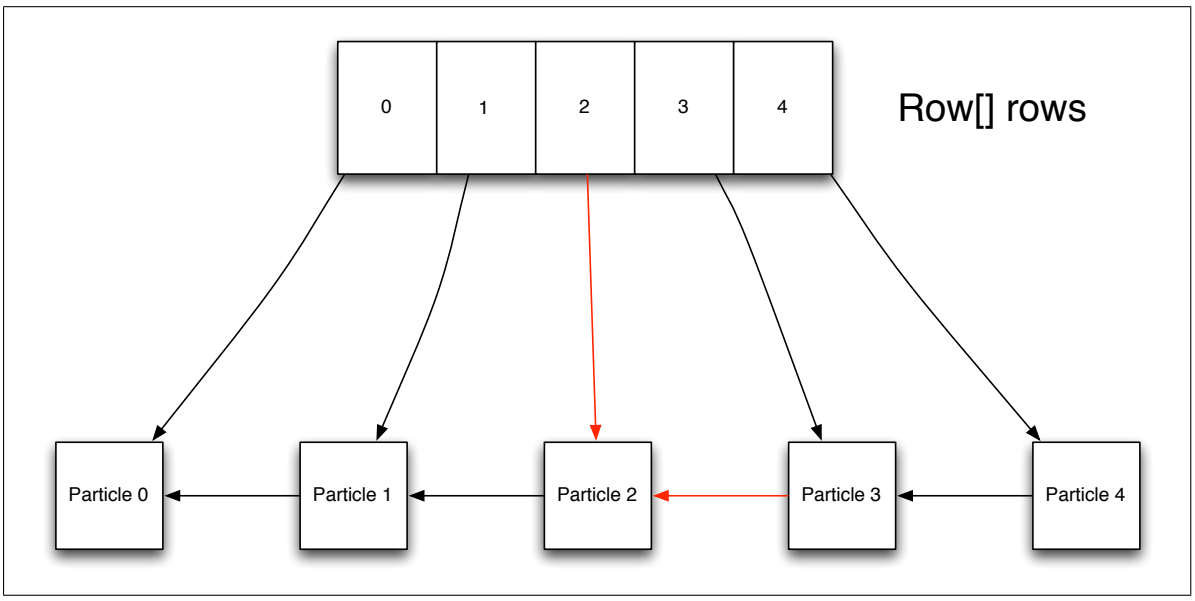

Figure 2.20: Memory cleanup: In order to remove particle 2, both references (from the array as well as particle 3) must be deleted. Otherwise, the garbage collector will not destroy the object.

```
// Step through the rows..
for (var i=0; i<rows.length; i++){
 // Step through each particle in the current row...
 for (var j=0; j<rows[i].particles.length; i++){
  if (rows[i].particles[j] != null){
   if (rows[i].particles[j].finished == true){
    // Remove the current particle if not needed anymore
    delete rows[i].particles[j];
   }
   else{
    // .. else keep moving it
    moveParticle(rows[i].particles[j]);
   }
  }
 }
}
```

The implementation of the test for **Adobe Flex** is similar to its JavaScript pendant, but as mentioned above there is no method like `clearRect()` provided in the API. Therefore, it is unavoidable to clean the whole stage using the function `canvas.graphics.clear()` on each iteration and then redraw all elements. This is not very efficient hence it is expected to cause frame-drops, compared to other implementations. Regarding the destruction of particle objects: There does exist a `delete` operator in ActionScript3 but this is usually not the best practice. Instead, the »Java way« of getting rid of unneeded objects is used, which requires the assistance of the Flash Garbage Collector. This subsystem of the Flash Runtime Environment removes all objects from memory if there is no variable or other object referencing this object. Thus, it is easy in this test to destroy

Class	Package (Filtered)	Cumulative Instances	Instances	Cumulative Memory	Memory
Particle	net.timoernst.riabench.test2d.model	5120 (94.4%)	0 (0.0%)	266240 (96.73%)	0 (0.0%)
Test2DDemo		1 (0.02%)	1 (6.67%)	1640 (0.6%)	1640 (19.07%)
Test2D	net.timoernst.riabench.test2d	1 (0.02%)	1 (6.67%)	1296 (0.47%)	1296 (15.07%)
TestCanvas	net.timoernst.riabench.test2d.view	1 (0.02%)	1 (6.67%)	1268 (0.46%)	1268 (14.74%)
Status	net.timoernst.riabench.test2d.view	1 (0.02%)	1 (6.67%)	1060 (0.39%)	1060 (12.33%)
Headline	net.timoernst.riabench.test2d.view	1 (0.02%)	1 (6.67%)	1060 (0.39%)	1060 (12.33%)
FpsLabel	net.timoernst.riabench.test2d.view	1 (0.02%)	1 (6.67%)	1060 (0.39%)	1060 (12.33%)
_Test2DDemo_mx_r		1 (0.02%)	1 (6.67%)	928 (0.34%)	928 (10.79%)
MethodQueueEleme	UIComponent.as$142	46 (0.85%)	0 (0.0%)	256 (0.09%)	0 (0.0%)
ChildConstraintInfo	CanvasLayout.as$454	4 (0.07%)	1 (6.67%)	192 (0.07%)	96 (1.12%)
ModuleManagerImpl	ModuleManager.as$26	92 (1.7%)	1 (6.67%)	60 (0.02%)	60 (0.7%)
LocaleID	LocaleSorter.as$107	4 (0.07%)	0 (0.0%)	40 (0.01%)	0 (0.0%)
TestController	net.timoernst.riabench.test2d.controll	1 (0.02%)	1 (6.67%)	32 (0.01%)	32 (0.37%)
en_US$styles_prope		1 (0.02%)	1 (6.67%)	20 (0.01%)	20 (0.23%)
en_US$skins_prope		1 (0.02%)	1 (6.67%)	20 (0.01%)	20 (0.23%)
en_US$effects_prop		1 (0.02%)	1 (6.67%)	20 (0.01%)	20 (0.23%)
en_US$core_proper		1 (0.02%)	1 (6.67%)	20 (0.01%)	20 (0.23%)
en_US$containers_p		1 (0.02%)	1 (6.67%)	20 (0.01%)	20 (0.23%)

Figure 2.21: Flex Builder's Memory Profiler: All instances of »Particle« have been destroyed by the Garbage Collector (marked red here).

the particles, which are not required anymore. First, the reference to its forerunner must be removed in order to prevent memory leaks:

```
rows[rowIndex][particleIndex].forerunner = null;
```

Then, the object itself must be removed from the array holding the rows and particles by calling:

```
// Leave particle to garbage collection
rows[rowIndex][particleIndex] = null;
```

This particle will not be removed from memory yet, but when the objects referencing this particle also get removed, so does their inner reference to this particle. Then the Garbage Collector can finally delete the object from memory. At the end, when all particles reached the bottom of the stage, all created instances were destroyed (see fig. 2.21 on page 75).

The implementation of the test for **Silverlight** is basically a mix of the JavaScript and Flex version. Microsoft's API provides a method to remove objects, like rectangles, explicitly by calling `canvas.Remove(myrect)`. Thus, it is easy to move objects from one point to another by simply first deleting it and then just redrawing it to its new position. No screen-redraw is required. In the same time, C# does not support the `delete` operator. Similar to its Flash-pendant, all references were set to null, which forces the built-in Garbage Collector to remove all redundant objects from memory. Since both techniques have already been explained in the prior paragraphs, detailed source code is omitted for the Silverlight implementation.

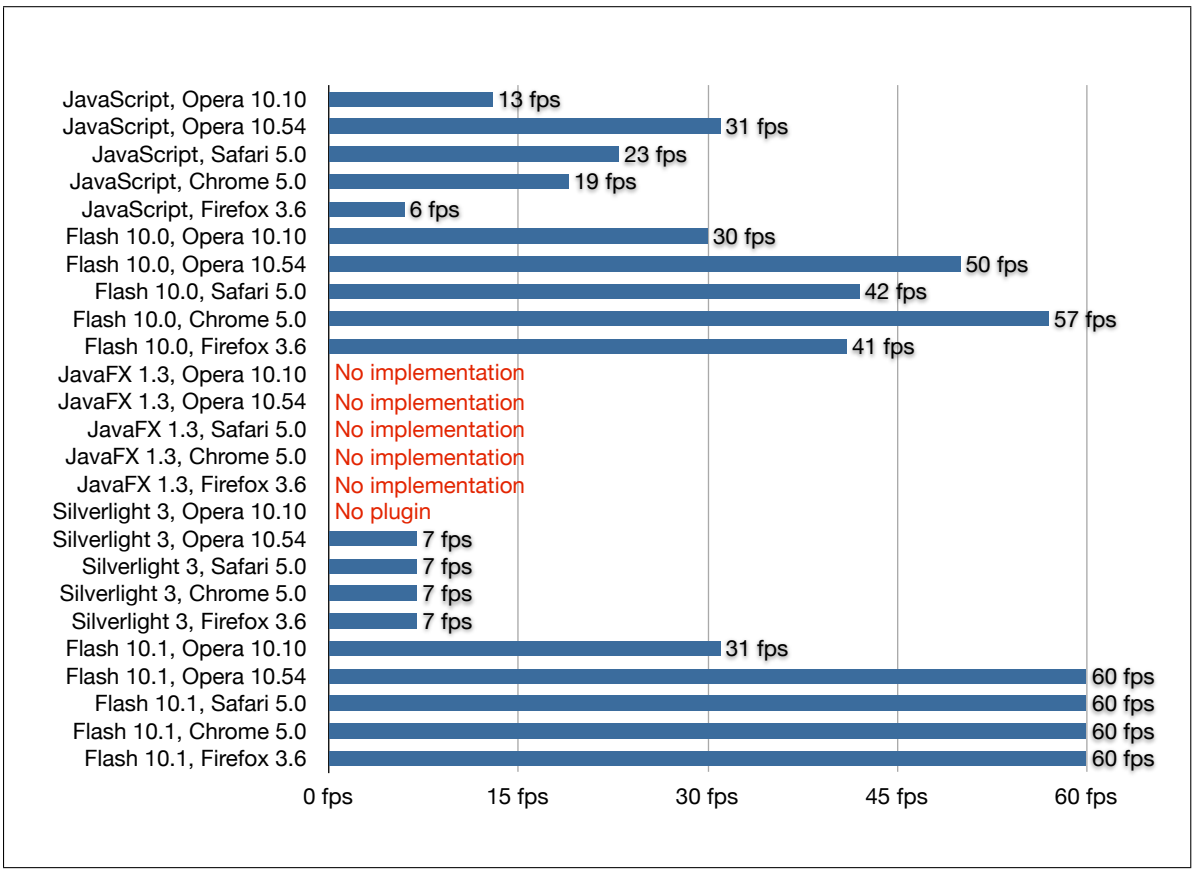

Figure 2.22: Results of the 2D-test on Mac OS X 10.6

Note: The test for **JavaFX** was not possible due the fact that there is no way to get the current fps rate. The reason for this is that there is simply no event which fires upon screen-refreshes occurring, hence there is no way to calculate the fps.

Results

(**Note:** Despite the results from other benchmarks, this test outputs fps values instead of milliseconds. Thus, lower numbers are better while higher ones are worse.)

Looking at the results from **Mac OS X**, it can be said that the Silverlight versions performed very badly at a constant rate of only 7 fps on all browsers together with the JavaScript-version of Firefox 3.6 (6 fps) while their Flash-pendants did very well, especially on version 10.1, running constantly at the desired 60 fps with one exception: Opera 10.10 seems to slow down the animation by almost 50% (see figure 2.22 on page 76). Currently, it is not clear why this happens but it is assumed that the cause for this anomaly is the browser's own rendering engine. In prior tests, it could be observed that Opera seems to interrupt running programs in order to refresh the screen. This behavior

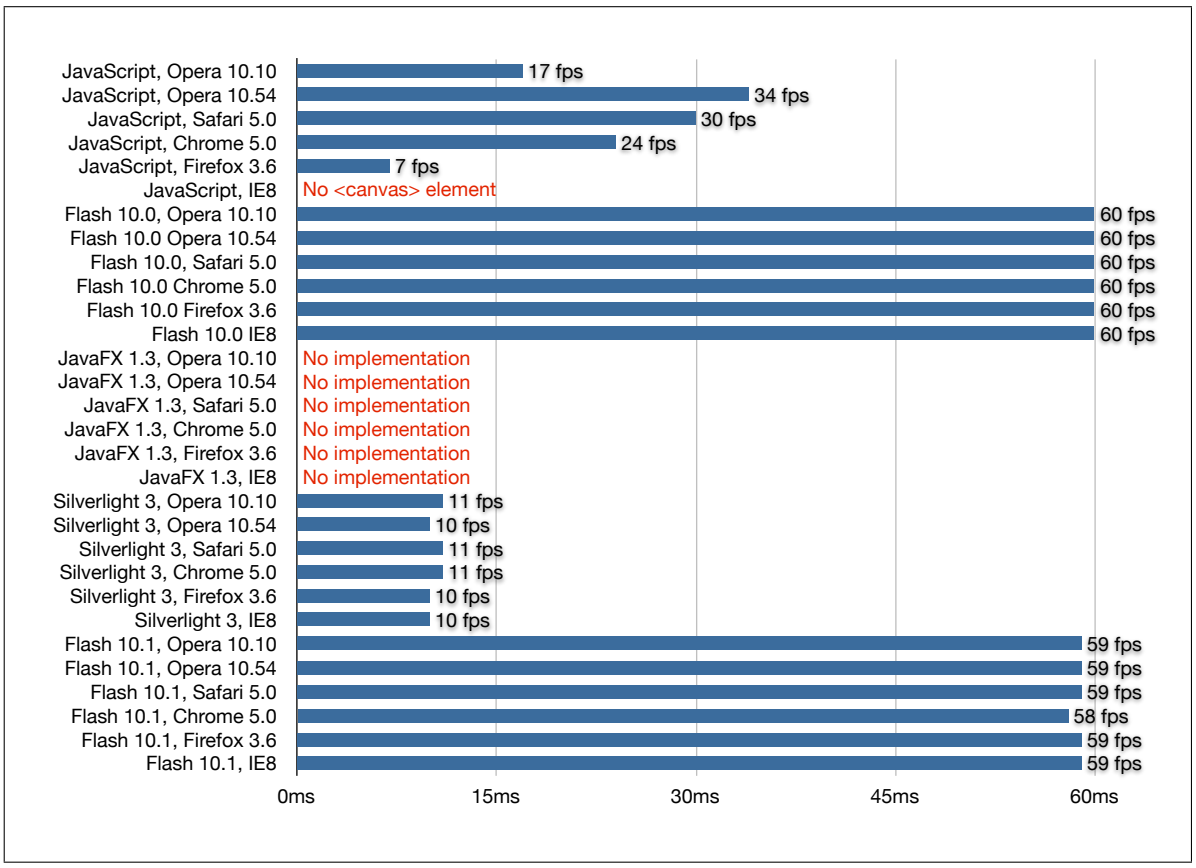

Figure 2.23: Results of the 2D-test on Microsoft Windows Vista

usually improves the user-experience when applications need to process a lot of data and the user cannot tell if the script has just frozen and will come back or if the browser must be force-closed. In this case, the interrupts caused by Opera seem to either interfere with Flash's own drawing engine or influence the fps measurement system of the benchmark. Since this was not investigated further in this thesis, results from this 2D-test must be treated with care. Regarding the JavaScript-versions of the test, it must be said that the distribution is partially unbalanced. The best performance here comes from Opera 10.54 at an average rate of 31 fps, followed by Safari 5.0 (23 fps), Google Chrome (19 fps) and Opera 10.10 (13 fps). Last place goes to Firefox with 6 fps. As mentioned in the prior section about the test setup, no measurement results for JavaFX could be achieved due to the fact that it was impossible to get the current fps rate of applications, created with this technology.

Regarding the test-results on **Microsoft Windows Vista**, it can be said that the most conspicuous finding was that the Flash Player 10.0 versions were actually a little bit faster than their successor 10.1 (see figure 2.23 on page 77). This observation matches with results from other test and leads to the conclusion that obviously the acceleration of Flash Player's new version on Windows was not as heavy as on Mac OS. Actually,

in some tests, like in this one, it could be shown that the performance increase was even negative. Silverlight as well as JavaScript on Firefox 3.6 and Opera 10.10 created the worst frame-rates between 7 and 17 fps. All other technologies made the jump over the »magical« mark of 21 fps. This is important since human beings start to recognize animations as being »smooth« at about 25 fps (as explained in section 2.3.1.3 about fps-rates and their impact on the human percipience) while a frame-rate of 21 fps leads to at least some frame-drops but is usually still tolerable.

On **Ubuntu Linux 10.04**, no additional obtrusive observations could be made, except for the fact that JavaScript on Firefox 3.6 as well as the Moonlight-version of this test were even slower (both at an average rate of only 1 fps) than compared to the Mac- and Windows-versions, which were already performing very badly there.

In summary, it can be said that Flash performed very well across all operating systems regarding 2D-animations, followed by JavaScript on Chrome 5.0, Safari 5.0 and Opera 10.54. Other browsers like Firefox 3.6 and Opera 10.10 could not catch up to their competitors. The last place goes to Silverlight/Moonlight, which performed very badly, no matter on which operating system or browser.

2.3.2.6 Memory-management and garbage-collection test

During the development of the Run Length Encoding test (described in section 2.3.2.4 on page 64), it could be observed that Mozilla's Firefox browser sometimes crashed during this benchmark throwing an out-of-memory exception. Furthermore, based on reports by Dipl.WiWi Steffen Fritzsche of the University of Ulm that web-browsers often tend to either crash or dramatically slow down in performance if huge numbers of JavaScript objects are involved some additional investigations were started. Based on these informations, a dedicated test for analyzing memory-usage and -handling was developed. This test builds on top of the 2D-benchmark as described in the prior section since that test already creates a lot of objects (of type »Particle«), so that it is not necessary to re-create algorithms for this purpose. Furthermore, the approach of »recycling« this prior test provides the ability to achieve results, which can be compared in order to see differences regarding performance based on the current and average values of fps. The basic idea of this test is to simply omit all lines of code which either explicitly remove objects from memory or set references to null in order to force the Garbage Collector to delete these objects next time it is being called.

Note: As mentioned in the section before, there is no 2D-test for JavaFX due to technical difficulties. Since this memory-test is based on the prior 2D-benchmark there is no such test for JavaFX. The implementations for JavaScript, Flex and Silverlight work as described.

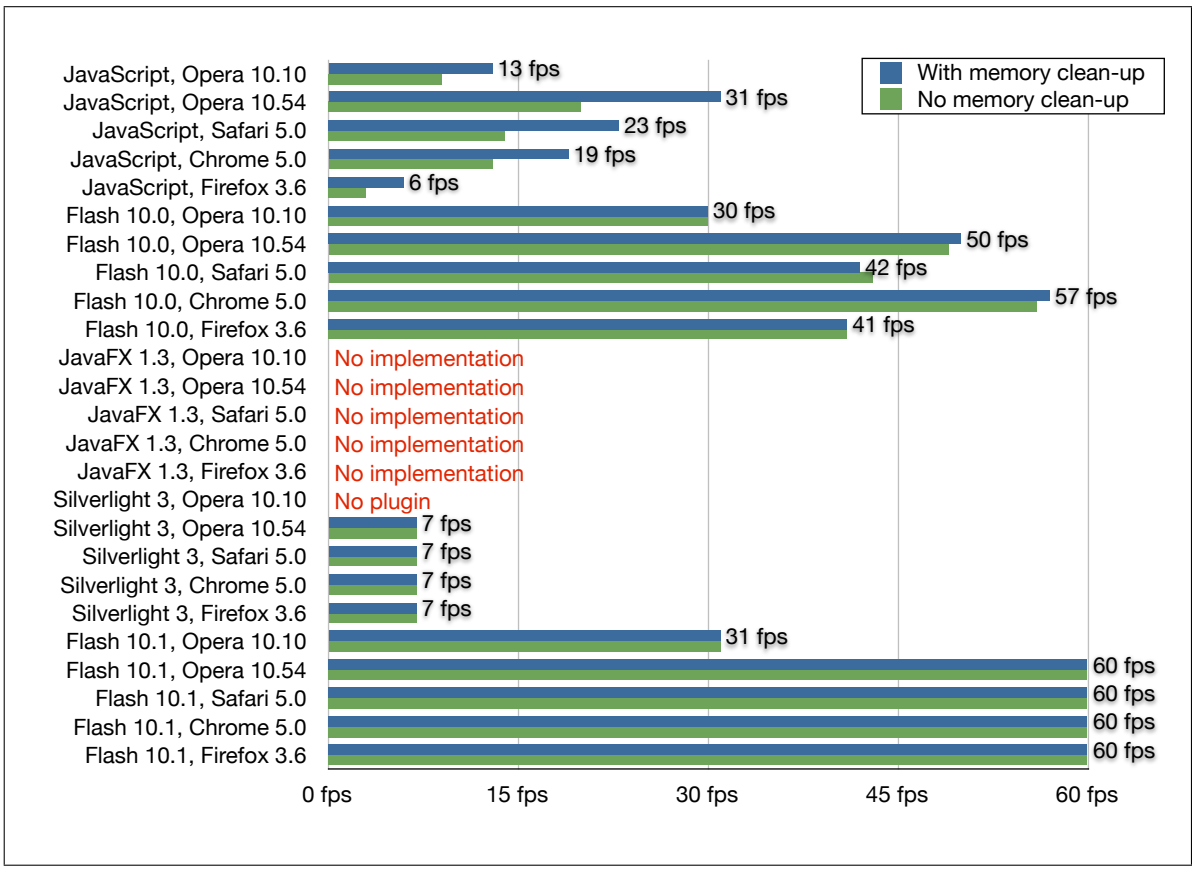

Figure 2.24: Results of the Memory-management-test on Mac OS X 10.6

Results

Looking at the results of this test, it could be observed that only the JavaScript-versions benefited from the explicit destruction of the particle-objects, which already reached the bottom of the canvas and were thus not necessary anymore **across all operating systems** (see figures 2.24 on page 79 for the results on Mac OS X for example — Results on Windows and Linux look similar). Neither Flash, nor Silverlight or JavaFX experienced a performance boost through freeing memory. This totally makes sense because, the total memory allocated by the sum of particle-objects was only around 266 kb (see figure 2.21 on page 75). Since the computer, these tests were running on, had 8 GB of RAM on 64-Bit Systems and around 3.5 GB on 32-Bit software available, no change in performance was expected. If this assumption is correct, then it must be asked why all JavaScript-versions were influenced by huge numbers of unused objects which were remaining in memory. Since this fact was not examined any further in detail, this question stays unanswered and requires additional investigation.

2.4 RIABench

2.4.1 Focus-tests

As mentioned in the introduction in this thesis, some benchmarks like Bubblemark or GUIMark test the performance of the platform they run on and output a result value either in milliseconds, fps[17] or benchmark-points. Other solutions like Google's JavaScript-only testing tool *V8* (see figure 2.25 on page 81) also include more use-cases, like crypto analysis or ray tracing algorithms and not only graphical testing. But again, these results are only a scratch to the surface and don't give any more detailed information regarding the reason of a specific result.

Because of this, and based on the previously developed API- as well as Non-API tests, the RIABench Focus-test was developed which takes all the insight from the prior measurements and compresses it into one single benchmark. Furthermore, no use-cases, like specific algorithms, were tested. Instead, the important parts were taken out of the use-case tests and explicitly re-implemented with as few as possible other interfering code. Thus, the following tests may look a bit simple, but they are the core of what should be tested regarding performance analysis on Rich Internet Applications. For example, many cryptographic algorithms are based on the (bitwise) XOR-operation. In order to implement this, functions which calculate $a \oplus b$ must be written. The XOR-operation returns 1 if a and b differ from each other and 0 if they are equal. Thus, it is easy to implement this as shown below:

```
function xor(a, b){
 if (a != b) return true;
 return false;
}
```

In order to write a test for this, »enough« runs need to be made calling this function over and over again with different parameters for a and b, but this is basically all that's needed. There is no reason why one should implement full cryptographic algorithms if the basic aspects, which are relevant for performance issues, can be sliced out. This way of benchmarking makes the implementation easier, gives a better overview over the results and covers a wide range of possible use-cases although they might be looking very simple.

All test implementations are wrapped by some lines which do the time measurement. The result of each test is stored into a dedicated variable. After all runs finished, their results are summed up and shown to the user. This way, the main result value is not being influenced by any non-relevant operations. The following code snippet shows the basic idea of the test setup in JavaScript:

[17] Frames per second

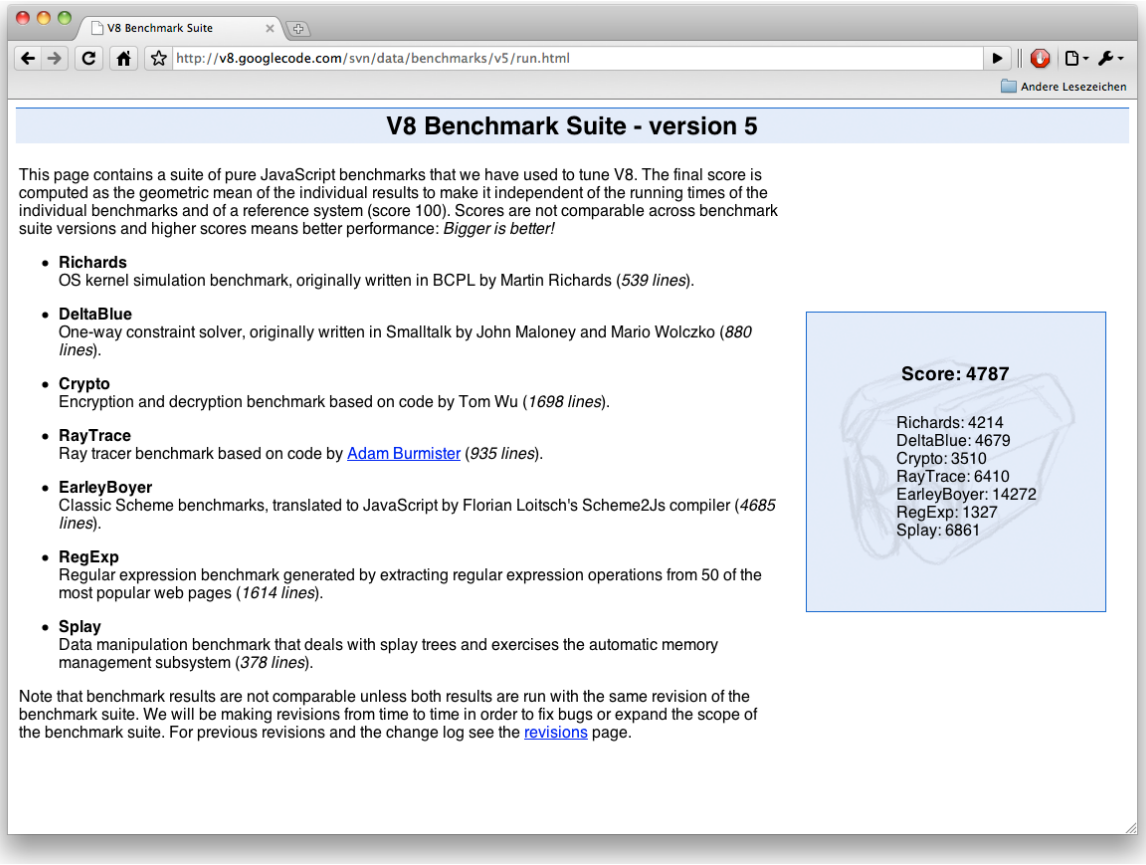

Figure 2.25: Google's V8 Benchmark for JavaScript

```
var globalTimeElapsed = 0;
// Redo this until all tests finished
while(moreTestsLeft){
 var startTime = new Date();
 startNextTest();
 var stopTime = new Date();
 var timeElapsed = stopTime.getTime() - startTime.getTime();
 globalTimeElapsed += timeElapsed;
}
outputResult(globalTimeElapsed);
```

Note: See *http://www.timo-ernst.net/riabench-start* for the full source for JavaFX, Silverlight, JavaScript and Flash.

As already mentioned before, application performance can either be influenced by the API library underneath or the execution runtime itself. Thus, both versions were implemented whenever possible. Since not every API-call in »A« has an equivalent in »B«, some tests might lack an API-version.

Overview

The whole benchmark is build on n so-called »Test-collections«, which are sets of small benchmarks dedicated for the investigation of special operations, like for example string-concatenation. These are called »Sub-tests«. The following table lists all Test-collections and their Sub-tests together with an unique identifier name so references in this thesis are easier to accomplish.

Sub-test name	**Test-collection**	**Info**
StringCharAtTest	StringTestCollection	Iterates over an existing string and pulls out each single character using the `str.charAt(i:int)` method
StringConcatTest	StringTestCollection	Concatenates two strings by using the (+)-operator
StringConcat-TestAPI	StringTestCollection	Concatenates two strings by using the `str.concat(str2)` API call
StringConcat-TestBuffer	StringTestCollection	Uses a StringBuffer for concatenation
StringConcat-SingleTest	StringTestCollection	Like StringConcatTest but with single characters instead of bigger chunks
StringConcat-SingleTestAPI	StringTestCollection	Like StringConcatTestAPI but with single characters instead of bigger chunks
StringConcat-SingleTestBuffer	StringTestCollection	Like StringConcatTestBuffer but with single characters instead of bigger chunks
StringGui-PushTest	StringTestCollection	Dumps a lot of text-data to a GUI-component (one by one)
StringIndexOf-Test	StringTestCollection	Searches for all letters of the latin alphabet inside a string (one by one) using an own implementation
StringIndexOf-TestAPI	StringTestCollection	Same like StringIndexOfTest, but uses the `str.indexOf(c:char)` API call
StringSubstrTest	StringTestCollection	Slices chunks of length 2 out of a given string using an own implementation
StringSubstrTest-API	StringTestCollection	Slices chunks of length 2 out of a given string using the `string.slice(from:int, to:int)` API call
ArrayIndexOfTest	ArrayTestCollection	Same like StringIndexOfTest, but with Arrays
ArrayItemAtTest	ArrayTestCollection	Same like StringIndexOfTestAPI, but with Arrays
ArraySubArray-Test	ArrayTestCollection	Same like StringSubstrTest, but with Arrays

Sub-test name	Test-collection	Info
ArraySubArray-TestAPI	ArrayTestCollection	Same like StringSubstrTestAPI, but with Arrays.
ArrayPopTest	ArrayTestCollection	Takes out the last element of an array and then removes it from the stack
BiggerThanTest	RelationalOperator-TestCollection	Compares two bits using the $>$ operator
EqualToTest	RelationalOperator-TestCollection	Compares two bits using the $==$ operator
SmallerThanTest	RelationalOperator-TestCollection	Compares two bits using the $<$ operator
MathAddition-Test	MathTestCollection	Calculates $a + b$
MathDivisionTest	MathTestCollection	Calculates a/b
MathModuloTest	MathTestCollection	Calculates $a \ mod \ b$
MathMulti-plicationTest	MathTestCollection	Calculates $a * b$
MathPowerOf-Test	MathTestCollection	Calculates x^n with an own implementation
MathPowerOf-TestAPI	MathTestCollection	Like MathPowerOfTest but uses the `Math.power(x, n)` API call
MathSqrtTest	MathTestCollection	Calculates `sqrt(a)`
MathSub-tractionTest	MathTestCollection	Calculates $a - b$

The whole benchmark was highly modularized. Programmatically spoken, every Test-collection must extend an abstract class called `TestCollection` and every Sub-test must extend `SubTest` (also abstract). This way, new tests can be added easily to the benchmark by extending the right class and then plugging in the module without having to rewrite existing code. Sub-tests are being plugged in to a Test-collection by calling `myTestCollection.addSubTest(mySubTest)` after implementing the method `startTest()`, which was declared by the abstract class `SubTest`. This method is being called every time a test should start to run. The time-measurement is being done automatically by the benchmark itself, so by implementing `startTest()` and hooking in the test, the programmer is done. If a new Test-collection is desired, a new class must be defined extending `TestCollection`, which must then be attached to the test-controller, which is usually simply called »TestController« (or similar, depending on the RIA runtime). This class basically does nothing more than iterating over all Test-collections and starting each test (see next page for the source-code).

```
1  // Will be called after each Test-collection finished and on
2  // program start
3  public function runNextTest():void {
4   if (currentTestIndex < testCollections.length) {
5    // "current" is a global variable holding the index number
6    // of the current Test-collection
7    testCollections[current].startTestCollection();
8    current++;
9   }
10  else {
11   allTestCollectionsFinished();
12  }
13 }
```

Listing 2.2: Partial implementation of the Test-controller written in AS3

The following sub-sections will give some more detailed informations for specific sub-tests of *RIABench*.

2.4.1.1 String operations

Based on the results of the RLE[18] test, a set of handling Strings was implemented, which includes the concatenation, separation as well as searching of characters. Furthermore, based on the result of the random key generator test, dumping strings to GUI components was also developed because JavaFX as well as Silverlight revealed some heavy weaknesses regarding this issue during the measurements. Thus, a test which inspects this problem in a smaller scope and isolates it from the rest of the algorithm was included to this set. In most programming languages, there are three ways of concatenating strings, which leads to three different tests:

- Using the (overloaded) (+)-operator
- Using a special concatenation-method provided by the API
- Using a string-buffer

Often, the (+)-operator is being overloaded in order to provide easy string concatenation. A simple example in pseudo-code would look like this:

```
var result = "Hello" + " world";
```

The variable »result« should now hold one single string containing the two words »Hello world«. Since it is sometimes not known, how the actual implementation for this over-

[18] Run-length-encoding

loaded operator works, testing this totally makes sense. Most modern API libraries provide another method though to concatenate strings programmatically using a dedicated function, which could look like this:

```
var str1 = "Hello";
var str2 = " world";
var result = str1.concat(str2);
```

In theory, there should be no reason why this approach is slower or faster than using the (+)-operator but in order to not miss anything, this test was also included to the series. As a last solution, some API's offer the possibility to use a so-called string-buffer, which should increase concatenation operations significantly. The idea is as follows: All strings which should be concatenated are being stored into a buffer (usually an array) one by one. Until the whole result-string is not required, no concatenation operation is being executed. Instead, the single »snippets« remain in the buffer. When the complete string is needed, all fields are being taken out of the array and then converted into a string using an API-function like e.g. `Array.join(delimiter:String)`. Although the number of concatenations is not being reduced using this technique, the `Array.join(delimiter:String)`-method, most API's provide, is usually very fast and should lead to a performance boost almost every time. A simple implementation in ActionScript3 syntax could look like this:

```
var str1:String = "Hello";
var str2:String = " world";
var buffer:Array = new Array();
buffer.push(str1);
buffer.push(str2);
var result:String = buffer.join("");
// => result = "Hello world"
```

Since Silverlight as well as JavaFX both provide an own `StringBuffer`-class, the algorithm above was only implemented in Flash and JavaScript. Furthermore, in order to see if the length of strings influences runtime performance, each of the three tests above are run twice:

- Concatenate strings of length 1 (= single characters)
- Concatenate strings with length > 1

In order to get some input data to work with, a plain ASCii text with some random loremipsum-text of 197kb size was used for working with the following test-cases.

Searching in strings

Most String-classes in modern programming languages provide a method like for example `mystring.indexOf(char c)`, which searches for the first occurrence of the character c inside the given string and returns its index number. In this test, all letters of the alphabet from a-z and A-Z were used as the parameter c which results in 52 iterations (Numbers were not included, since none of these appear in the input text anyway). In order to be able to see if maybe an own, self-written version might be faster, an API- as well as Non-API version were developed.

(The indexOf-function was used in the RLE-encoding process in section 2.3.2.4 on page 64 in order to find the number of redundancies inside given strings. Thus, it is interesting to see how this particular operation performs in the Focus-test.)

Accessing single characters in Strings

In order to verify how fast each RIA solution can access single characters inside strings, a simple API-test was implemented using the `mystring.charAt(i:int)`-method most languages provide.

(This method also was used in the RLE-encoding process in section 2.3.2.4 on page 64 in order to find the number of redundancies inside given strings.)

Separating strings into sub-strings

The substring-function was used in the RLE-encoding process in section 2.3.2.4 on page 64 in order to cut out occurring redundancies. Since the substring-method takes quite a while to compute, only substrings in 2-character-chunks were taken. Otherwise the whole benchmark would take hours to complete. Since this implementation is done the same way for other RIA platforms, the results are representative.

(This method also was used in the RLE-encoding process in section 2.3.2.4 on page 64 in order to find the number of redundancies inside given strings.)

String-dumping to GUI-components

The random-keygenerator test for JavaFX and Silverlight has shown that dumping a lot of ASCii data to the UI[19] can become a big issue. Thus, this test was implemented, which outputs the content of a string-object step by step to a text container. In order

[19] User-interface

to limit the execution time, only $\frac{1}{50}$ of the original input string is being used. This type of data output is often required to show various information to the user. In the case of the key generator, all generated random values should be dumped.

2.4.1.2 Array operations

Arrays are used in almost all tests in this thesis and they are often an important factor regarding performance issues in general. Examples are the random key generator (see section 2.3.2.3 on page 62) where each value is being stored to such a stack (and has then to be searched and retrieved again) or the 3D-test (see section 2.3.1.3 on page 43) which stores each planet-object inside an array. If such applications heavily rely on array-operations, an effective implementation is absolutely vital for a good user-experience, otherwise performance losses can be the consequence.

Test setup

The set of tests for arrays is pretty much the same as the ones for strings, it only lacks a test which outputs characters to a UI component since this issue has already been covered in the previous chapter. Since not all API's provide these methods, they were implemented manually as described below (AS3[20] syntax). In order to have an input array to work with, the lorem-ipsum string from the previous section was converted from a string-object to an array.

Searching in arrays

The following test simulates the behavior of the `indexOf(char c)` method for strings. A given input array will be examined for occurrences of all letters from the latin alphabet. If a matching was found, the next letter will be used. The operation which is relevant for this testis the possibility to access each single element through the [ ]-operator. Detailed tests regarding the `==` operator using `if`-statements are being introduced in section 2.4.1.4 on page 89.

Accessing elements in arrays

Since the access of single elements is already natively implemented in arrays on all platforms using the [ ]-operator, a dedicated `myarray.charAt(i)` is not necessary (un-

[20] ActionScript 3

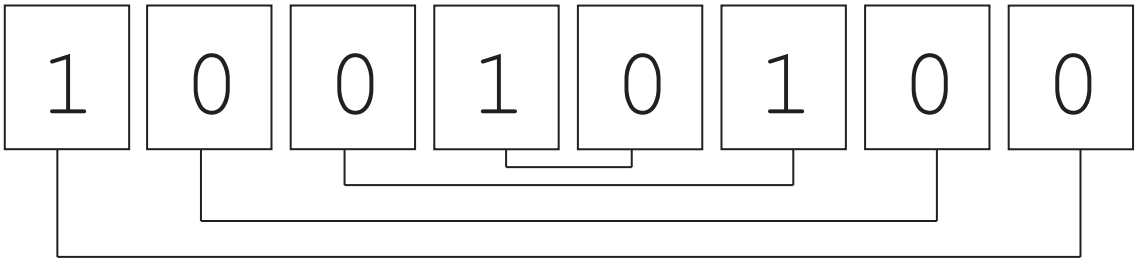

Figure 2.26: Comparison of single elements in a bitstream-string

like compared to strings). In order to verify, how fast each item can be accessed, the implemented test simply pulls out each character out of the input array.

Array-splitting

Strings often provide a method like `mystring.substr(start:int, end:int):String` to slice out specific parts. This is sometimes useful in combination with concatenation purposes in order to delete parts of an array. The downside of these, for example compared to linear lists, is that these operations have linear runtime complexity ($O(n)$) since the worst case scenario is that all elements of one (sub-)array need to be accessed and copied one by one. Compared to this, data structures like linear lists can easily handle these kind of operations by simply modifying pointers and thus have constant runtime complexity $O(1)$. Because of this, it is even more important that array operations, as shown below, are being implemented efficiently. The implemented tests (API- and Non-API version) verifiy how specific RIA platforms can handle these kind of problems by slicing out chunks of length 100 of the given input array.

2.4.1.3 Math operations

Mathematical operations are often vital for many kinds of software. Thus, it is important that these scale well regarding performance if it comes up to processing lots of input data. For example, the prime number test, introduced in a prior test in this thesis, relies on the modulo operation in order to decide if a number n is prime or not. The prime factorization test requires the `Math.sqrt(n)` method to determine when the algorithm needs to terminate. These are only a few examples where math operations are directly important, but there are also use-cases where these basic math functions play an indirect role. The 3D-test for example relies on the possibility to do matrix-multiplications.

Having a look at the way how this is being done, it becomes clear that it is not necessary to create a dedicated test for matrix multiplication because all that's required is the

addition and multiplication of single numbers:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{pmatrix}$$

In order to achieve a representative set of benchmarks, the basic math operation tests below were developed. As a source for (random) input numbers, the lorem-ipsum text from prior array-tests was converted into a sequence of integers by pushing the unicode-value of each element into a new stack:

```
var numbers = new Array();
for (var i=0; i<input.length; i++){
 numbers.push(input.charCodeAt(i));
}
```

2.4.1.4 Relational operators

The relational operators used in this set of tests are `==`, `>` and `<` using the `if`-statement which exist in all programming languages of RIA platforms introduced in this thesis. These operators represent the basis for many algorithms, especially in technical and cryptographic context. For example, many security-related techniques (like MD5-hashing, which is introduced in section 2.3.1.2 on page 40) rely on the XOR-operation ($\oplus$), which is defined as shown below:

a	**b**	$a \oplus b$
0	0	0
0	1	1
1	0	1
1	1	0

In simple terms, $a \oplus b$ always returns 1 `if (a != b)` applies. Thus, it is clear that the (`!=`)-operator is vital for the XOR-operation and hence plays an important role in cryptographic context among other algorithms.

The basic idea for all of the tests described here is to compare single bits inside a bitstream, which was achieved by converting the original lore-ipsum input file to a sequence of bits as shown as in listing 5.7 on page 129. Each element e_1 is being compared with another one e_2 inside a string of bits. If the index position of e_1 was 5, then the position of e_2 is bitstream.length-6 (See figure 2.26 on page 88). In order to not make things more complicated than necessary, a second, reverted version of the input bitstream was created using the code snipped shown in listing 5.8 on page 129. Now, it is possible to use a

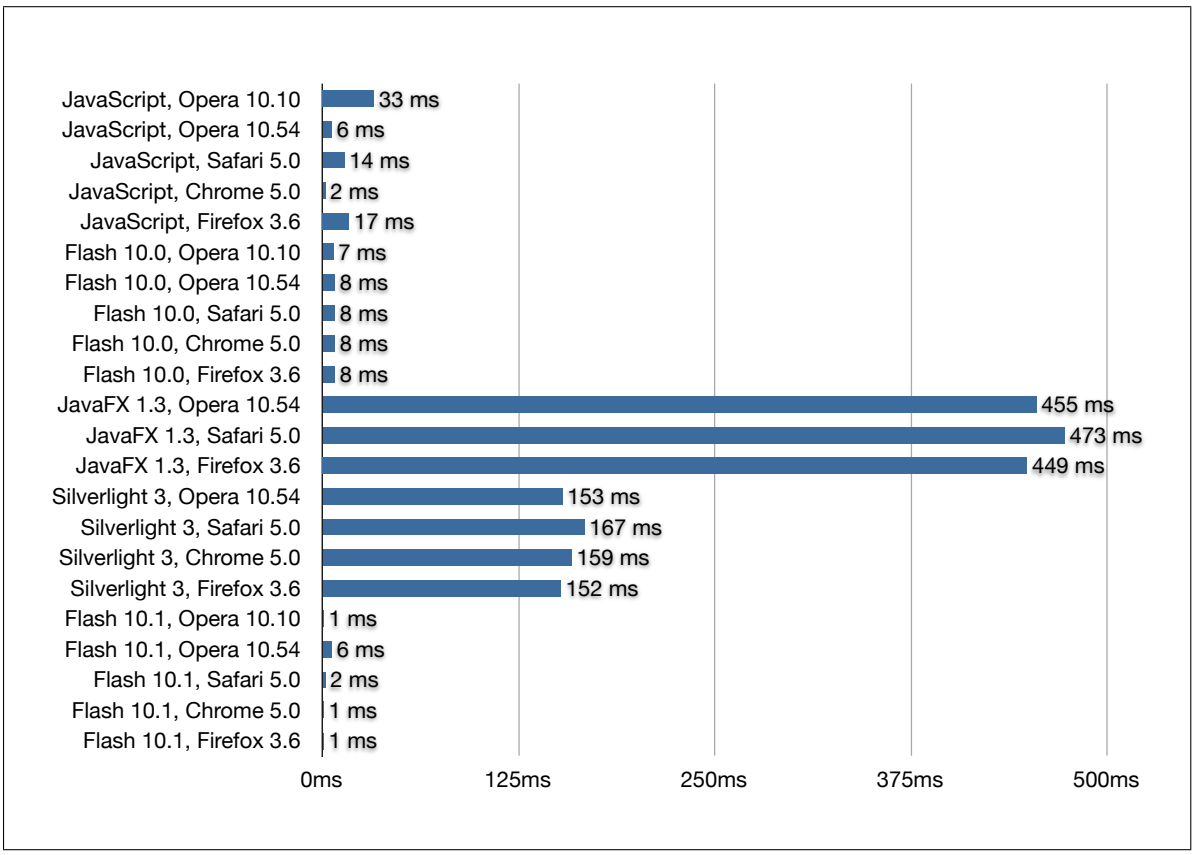

Figure 2.27: Result of the »StringConcatAPI« focus-test on Mac OS X 10.6

`for`-loop and compare each element of bitstream with its pendant in bitstream_reverted at the same index position per iteration step.

2.4.2 Results (Summary)

Regarding the results from the string-concatenation tests on **Mac OS X 10.6**, **Microsoft Windows Vista** and **Ubuntu Linux 10.04** it could be observed that all browsers running the JavaScript-version, except for Opera 10.10, performed superior to other solutions like JavaFX or Silverlight in most string-based benchmarks, especially the ones where the concatenation was accomplished through the (+)-operator as well as the `str1.concat(str2)`-methods (see figure 2.27 on page 90. The only exception where the so called integrated String-Buffer classes, which are built into the API's of JavaFX and Silverlight. If these were used, the string concatenation test finished in only 1-2ms (see figure 2.28 on page 91). The big winner regarding these operations is Flash in both versions 10.0 and 10.1 though because Adobe's plug-in was able to compute all concatenations, no matter how, in less than 5ms. This result cannot compete with JavaFX's and Silverlight's String-Buffer values but they simply were the most consistant. Regarding

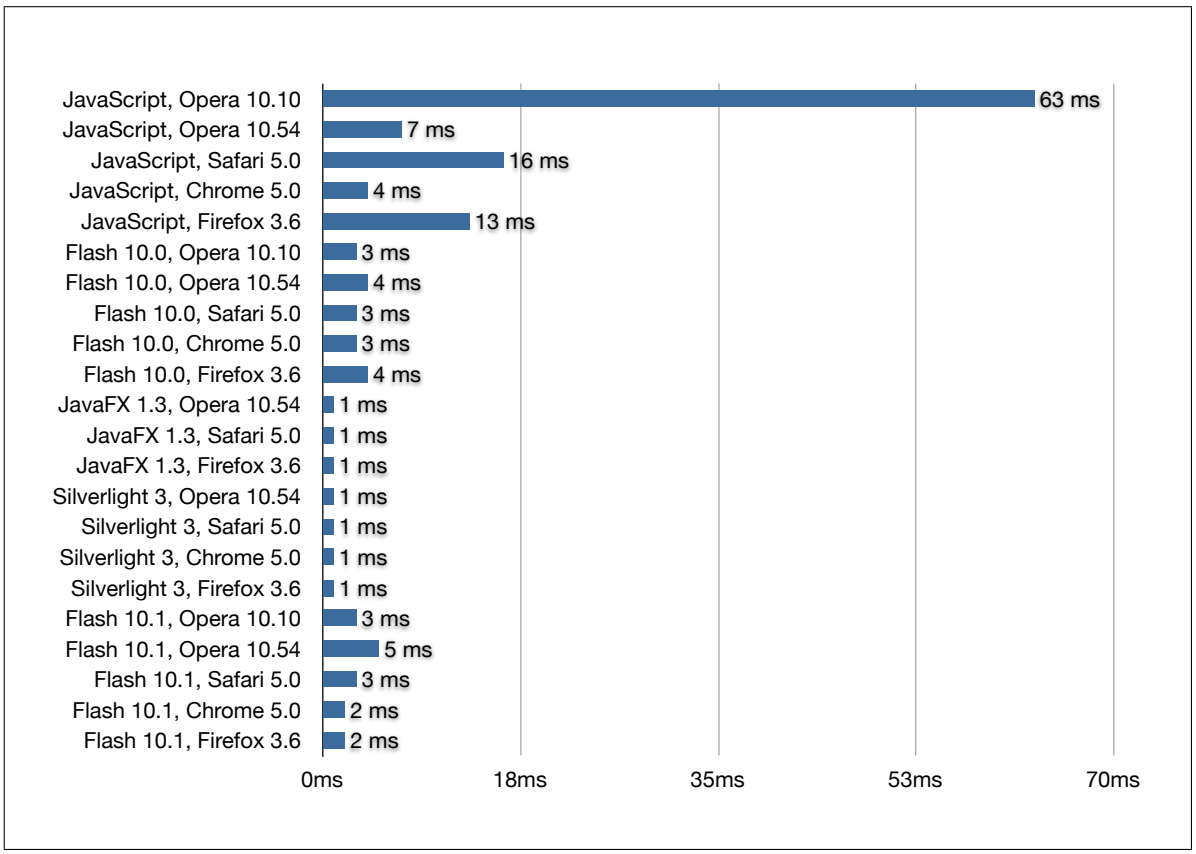

Figure 2.28: Result of the »StringConcatBuffer« focus-test on Mac OS X 10.6

the question which method should be used on which platform, the following suggestions can be made:

- For JavaScript, as well as Flash applications, the (+)-operator should be used for string-concatenation. The built-in `str1.concat(str2)`-methods should be avoided since most of these API-calls performed worse than the versions utilizing the (+)-operator.

- On JavaFX and Silverlight, the StringBuffer-class is a must. Using the (+)-operator or any other method to concatenate strings, will result in extremely slow application performance.

Regarding operations in strings using the method `str.charAt(i:int)` or the `[]`-operator (on Silverlight), it can be said that on **Mac OS X**, JavaFX as well as Flash 10.1 performed best with average rates of 2.7ms and 3.6ms (see figure 2.29 on page 92). The worst results could be observed on Flash 10.0 (all browsers) at 26-28ms and Opera 10.10 at 197ms (JavaScript) although it must be said that all other browsers did very well on JavaScript and finished the test in 5-10ms. On **Microsoft Windows Vista**, the results

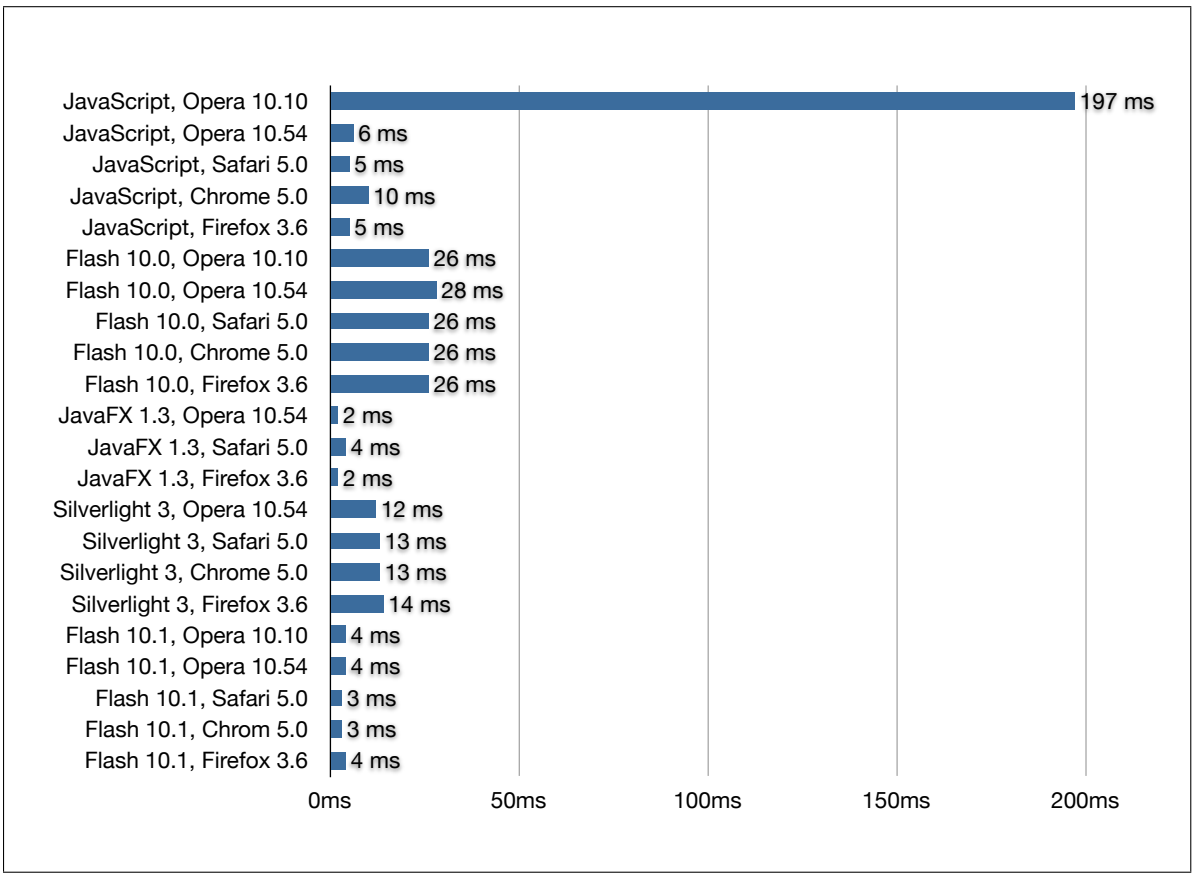

Figure 2.29: Result of the »StringCharAt« focus-test on Mac OS X 10.6

were similar: Opera 10.10 as well as Internet Explorer 8.0 produced the worst results (by far) at 146ms and 226ms while all other browsers did fine on the JavaScript-test at usually around 4-10ms. JavaFX again performed best, like on Mac OS X, followed by both Flash versions 10.0 and 10.1 since their results were pretty much equal at rates between 4-6ms. Silverlight is in average of the field at 35-38ms. On **Ubuntu Linux 10.04**, pretty much the same conclusions could be made as on Windows Vista.

The so called »GUIPushTest«, which dumps a lot of string values one by one to a GUI-element, like a TextBox or `<div>` element, showed that on **Mac OS X**, all browsers running the JavaScript-version, performed worst at rates between 227ms and 428ms, followed by Silverlight at an average value of about 178-184ms. The second place goes to JavaFX as well as Flash 10.0. The big winner of the test is Adobe's new version of the Flash Player (version 10.1) with only 14-16ms. On **Microsoft Windows Vista**, the results were almost identical but with one exception: Flash Player 10.0 was as fast as its successor 10.1 (see figure 2.30 on page 93). All other results are basically similar to the ones from Mac OS X. On **Ubuntu Linux 10.04**, it could be observed that Moonlight seemed to have serious problems with this test since the result of 810ms were definitely the worst in the field. All other technologies basically behaved liked on Mac OS and

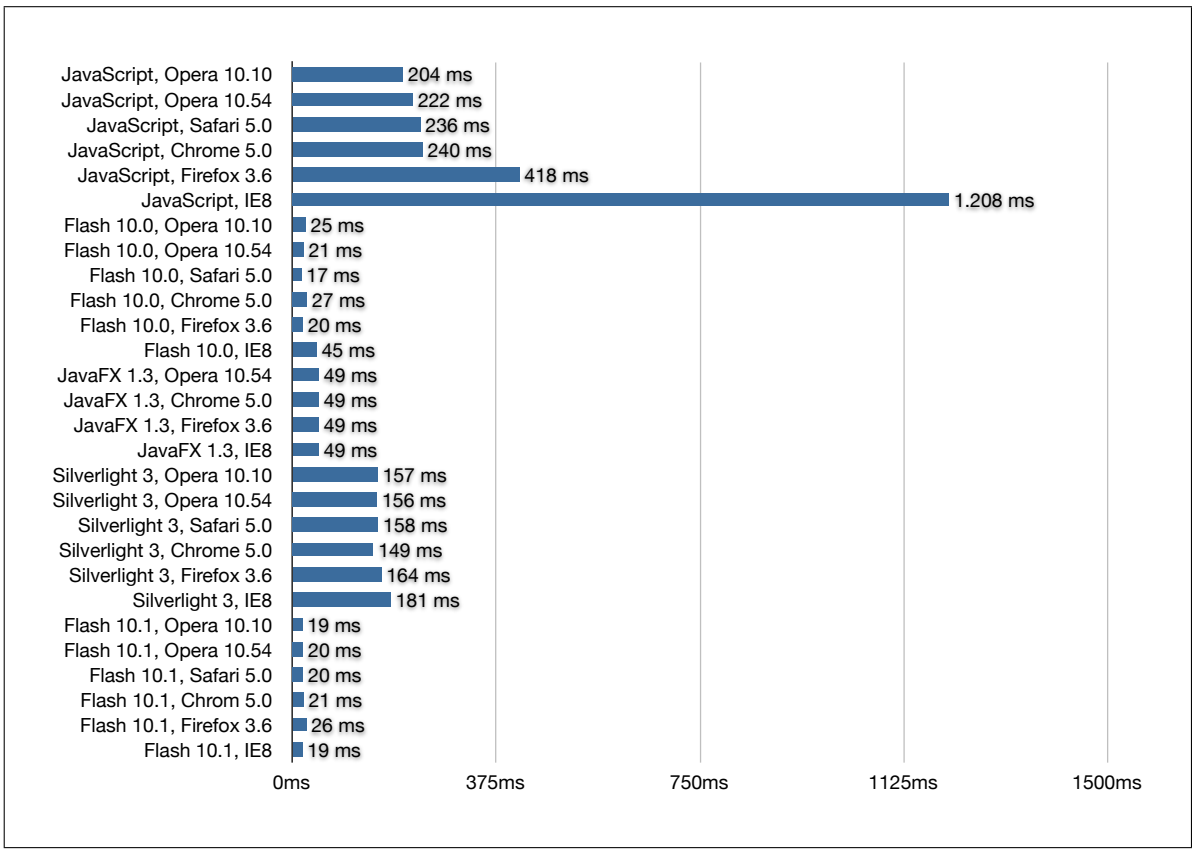

Figure 2.30: Result of the »StringGuiPush« focus-test on Microsoft Windows Vista

Windows: Flash 10.0 and 10.1 returned the best results, followed by JavaFX and then JavaScript. In summary, it can be said that it is no good idea to dump single strings one by one to the screen using JavaScript or Silverlight. Instead, the data, which should be displayed, must be cached into a string-buffer and then dumped in bigger chunks at regular intervals. On other technologies like Flash or JavaFx, these string-dumping should be quicker, but it is still recommended to use a buffer since user-experience can quickly drop with slow reacting user-interfaces.

If one wanted to search for specific characters inside strings, the `str.indexOf(c:char)` is probably the way to go. This approach is called the »API-version« here. Besides this method it is also possible to use string's `str.charAt(i:int)` function and iterate over each character in the string and check whether the desired letter or number can be found (the Non-API-version). Having a look at the results, it can be definitely said that the API-version was a lot faster than the Non-API variant. This result was expected and thus not very astonishing. A more detailed analysis between the single technologies showed that Silverlight/Moonlight performed with the best values at 3-21ms across all operating systems. All other runtimes (JavaFX, Silverlight and Flash 10.0) were pretty much equal at rates between 65ms (JavaScript, Opera 10.54 on Windows) and 199ms (JavaScript, Chrome 5.0 on Windows) except for Opera 10.10 and Internet Explorer 8.0

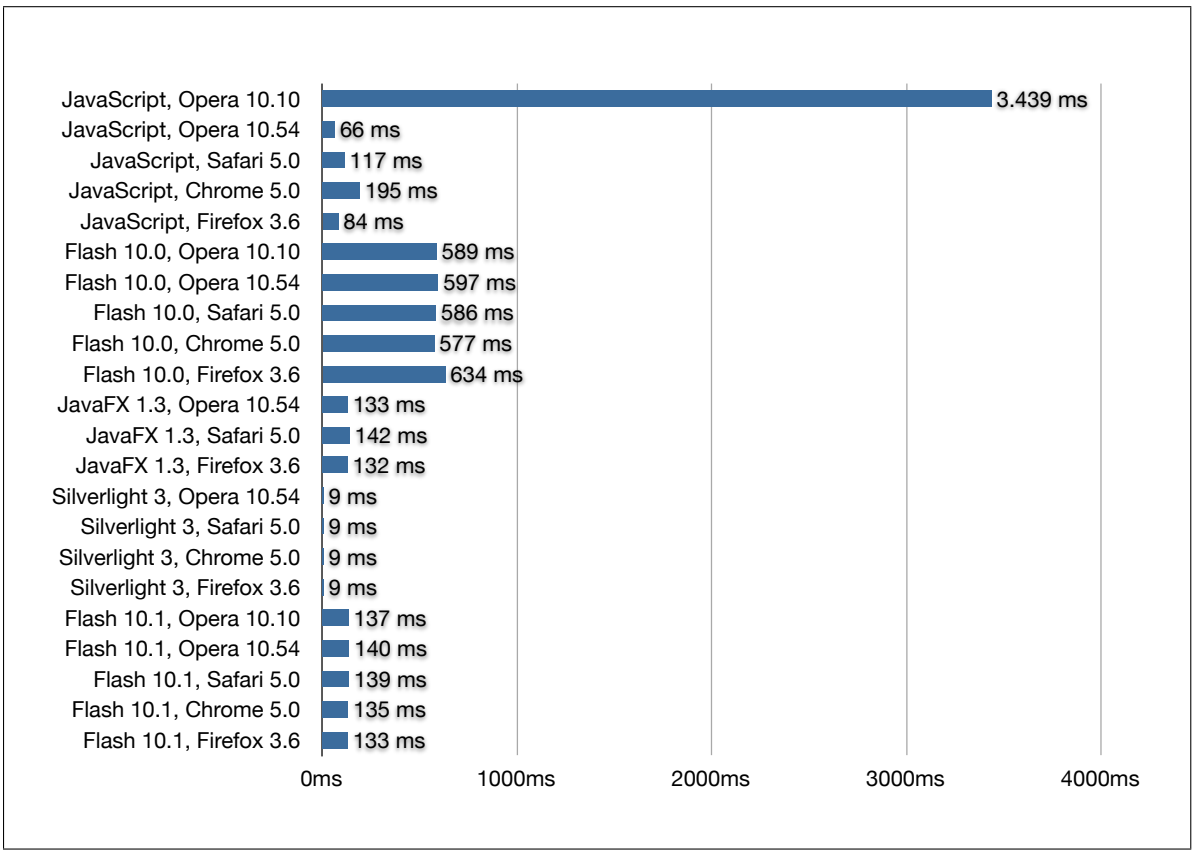

Figure 2.31: Result of the »StringIndexOf« focus-test on Mac OS X 10.6

which returned horribly low rates 2000ms-3500ms, depending on the operating system they ran on. On Mac OS, Flash Player 10.0 was about 5 times slower than compared to version 10.1 (see 2.31 on page 94). On Windows and Linux, no significant performance improvement could be observed.

The last string related test is the so called »StringSubstrTest« which cuts out chunks of characters out of a given string, which are defined through parameters a and b. For example the method `str.substr(a, b)` would return »abc« with str="abcdef",a=0 and b=3. Similar to the previous test, an API- as well as a Non-API version was developed. See `http://www.timo-ernst.net/riabench-start` for the full source code to download. Regarding the results, it must be said that the API-version was always superior to its Non-API pendant. Obviously, the manufacturers of the runtimes did some improvements to their API-functions, which lead to these performance benefits. Having a look at the results, it could be observed that JavaFX as well as Silverlight were both the fastest runtimes. If both extremely slow browser Opera 10.10 and Internet Explorer 8.0 were excluded, most JavaScript-engines can compete with these results though. Regarding Adobe Flash, it can be said that both, Flash 10.0 and 10.1 were about 2 times slower than their competitors, on Windows on Linux. On Mac OS, Flash 10.0 was about 13 times slower and 10.1 about 5 times. In summary, it can be said that cutting out sub-

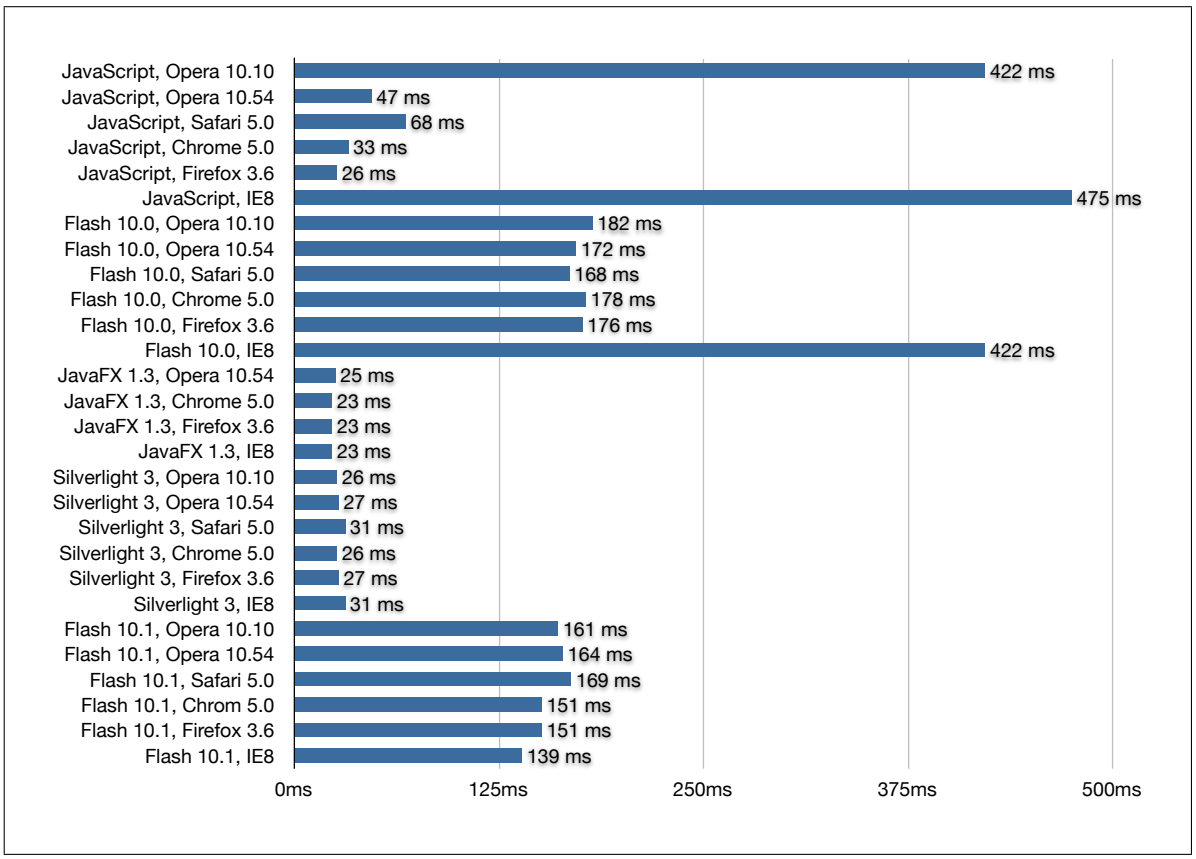

Figure 2.32: Result of the »StringSubstr« focus-test on Microsoft Windows Vista

strings works very well on JavaFX and Silverlight but also in JavaScript if neither Opera 10.10 not Internet Explorer 8.0 was used. Flash in both versions tended to be a little bit slow but still acceptable. Besides these results, the most astonishing thing observed was that the Internet Explorer 8.0 was more than two times slower than compared to other browsers on Flash 10.0 for the Non-API version (see figure 2.32 on page 95). Currently, it is unknown how this could happen, but obviously this can be considered as another proof that browsers can influence performance of applications which run inside a plug-in.

Looking at the first test which is related to operations on arrays, the »ArrayIndexOfTest« searches for specific items inside such structures and returns the index position, similar to the previously mentioned »StringIndexOfTest«. The results show that JavaFX as well as the JavaScript-versions (Opera 10.10 and Internet Explorer 8.0 excluded) performed best across all operating systems, followed by Flash and then Silverlight. On Mac OS X, Flash version 10.0 was about 25% slower than 10.1. On Windows and Linux, both were equal.

One of the most important functions for arrays is to get items out of them by specifying a specific index. This operation is usually done through the `[]`-operator, e.g.: `myarray[3]`

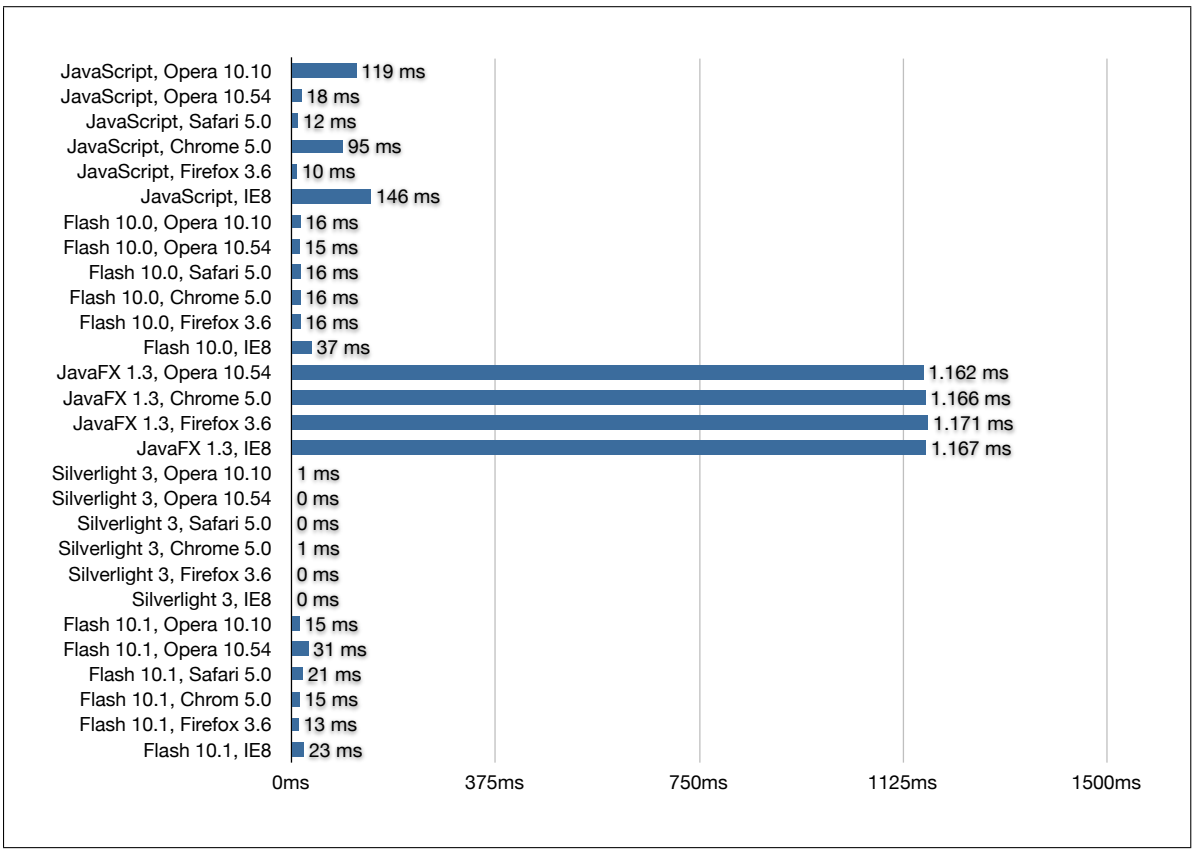

Figure 2.33: Result of the »ArraySubarray« focus-test on Microsoft Windows Vista

returns the element which is stored at index 3 in the array. The test results show that JavaFX performed best across all operating systems, together with JavaScript. Opera 10.10 and Internet Explorer 8.0 excluded, since they resulted in ridiculous slow rates. Opera 10.10 for example was 17 times slower than its competitor Google Chrome and the Internet Explorer 8.0 even 38 times slower. Place three goes to Silverlight on Windows and Mac OS. On Linux, it could be observed that Moonlight was about 3-6 times slower than compared to other operating systems.

The next test related to array is the so called »ArraySubarrayTest«, which does same to arrays what the previously discussed »StringSubstringTest« did to strings: Cutting out chunks of data. First the results of the non-API version are being analysed: Besides the fact that both, Opera 10.10 and Internet Explorer 8.0 again performed pretty badly, the absolutely most astonishing and not expected results was that JavaFX ended up with horribly low results, which are even way lower than the ones of the two previously mentioned browsers (see figure 2.33 on page 96). Obviously, it is a huge problem for JavaFX to iterate over such an array and cut chunks out. Looking at the results of the API-version, it becomes clear that this must be somehow associated to the way, the non-API version was implemented since the built-in operations for the so called array-slicing performed quite well on JavaFX if compared to other results. The code above shows the

way the non-API version was implemented:

```
function slice(arrayInput:String[], from:Integer, dest:Integer):String
    []{
 var res:String[] = [];
 for (i in [from..dest]) {
  insert arrayInput[i] into res;
 }
 return res;
}
```

Obviously, there can be only two causes for the huge performance loss on JavaFX: Either the iteration over the array itself is to slow, or the insert-operations shown here, which fills the temporary result-array, takes up to much time. Since for-loops over sequences were already done multiple times and no specific problems were noticed, it is very possible that the insert-operation is the cause for the performance loss, which could be observed here. Until further investigation, this is just an assumption though. For the moment, it can only be recommended to use the built-in notation for array-slicing in arrays like this:

```
res = arrayInput[i - 100..i];
```

.. instead of:

```
res = slice(arrayInput, i-100, i);
```

Regarding the last test on arrays about »popping« out the last element out of a stack (the so called »ArrayPopTest«), no specific abnormalities could be observed. Most runtimes performed well between 5-12ms across all operating systems (except for Opera 10.10 and Internet Explorer 8.0, which needed 36ms and 121ms on Microsoft Windows). Only Flash 10.0 was a bit behind on Mac OS but this was also no big surprise since it could already be observed in prior tests that Flash 10.0 on Mac always behaves a bit slowly than compared to 10.1.

The next collection of test-series is the so called »RelationalOperatorTestCollection« which includes three tests:

- The »BiggerThanTest« using the $>$ operator
- The »EqualToTest« using the $==$ operator
- The »SmallerThanTest« using the $<$ operator

Looking at the results, it can be said that no significant difference between these three operators could be observed (see figure 2.35 on page 99 for an example chart on Linux). The only exception is the »EqualToTest« for JavaFX on Linux, which seems to be about

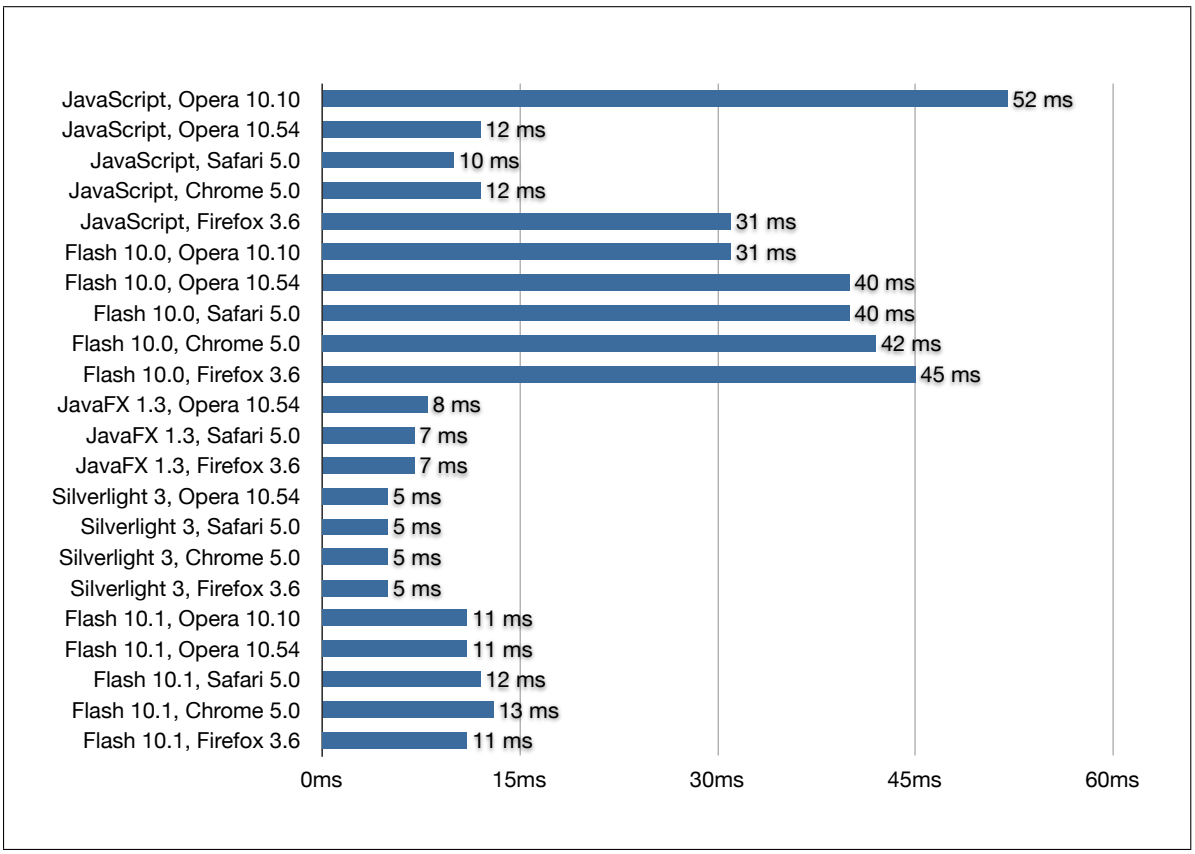

Figure 2.34: Result of the »ArrayPop« focus-test on Mac OS X 10.6

9 times slower than the values. Since this anomaly could not be observed in other tests, it is very possible that this result was caused through a measurement error because there is no reason why only this single value should return a result which differs that much from all other numbers. Thus, it was decided to sum up all results of this test-collection and then compare them among each RIA runtime and across the various operating systems. Looking at these results, it could be observed that JavaFX seems to be superior to all other RIA technologies, no matter on which o.s.[21] at rates between 31-90ms. Only Silverlight on Windows (47-52ms) and Linux (16ms, Moonlight with Firefox 3.6) could compete with these values. The Mac-version of Silverlight was significantly slower at 324-346ms. The next fastest results come from Flash 10.1 at 215ms (lowest on Mac OS) to 265ms (peak value on Ubuntu), which was about 2 times faster than Flash 10.0 on Windows and Linux and more than 3 times faster on Mac OS. JavaScript kept up at similar rates except for Opera 10.10 (3437ms-4558ms) and Internet Explorer 8.0 (7406ms) having the biggest performance loss. A significant impact on performance fluctuations, caused by the wrapping browsers the applications run in, could not be observed in these tests.

[21] Operating system

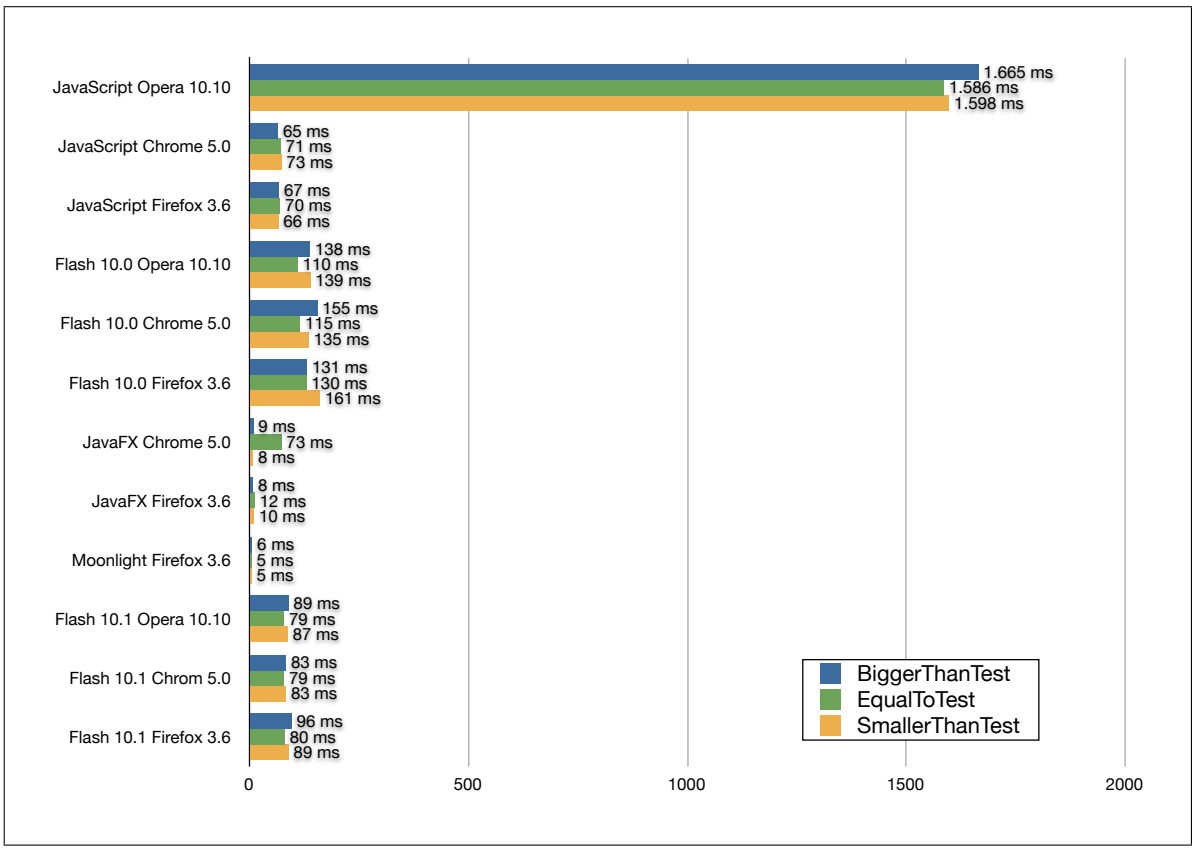

Figure 2.35: Results of the »RelationalOperatorTestCollection« on Ubuntu 10.04

The »MathTestCollection«, which is the last series in this benchmark, tests various mathematical operations like:

- Operators (+), (-), (*) and (/) for integers
- Modulo-operation
- Square-root calculation
- $f(x) = x^n$ (API- and non-API version)

Looking at the results from **Mac OS X 10.6**, it can be said that JavaFX and Silverlight are the top runtimes for doing mathematical calculations (see figure 2.36 on page 100), followed by Flash 10.1 on all browsers and the JavaScript-versions on Opera 10.54 (166ms) and Firefox 3.6 (137ms). It is astonishing to see how Mozilla's browser takes the lead for such math operations although it didn't perform that good in prior tests. Especially the Webkit-based browsers Chrome 5.0 (354ms) and Safari 5.0 (269ms) didn't do well here. The reason for this anomaly is simple: In 7 of 8 math-tests, all browsers (except for Opera 10.10) share almost the same results apart from some mi-

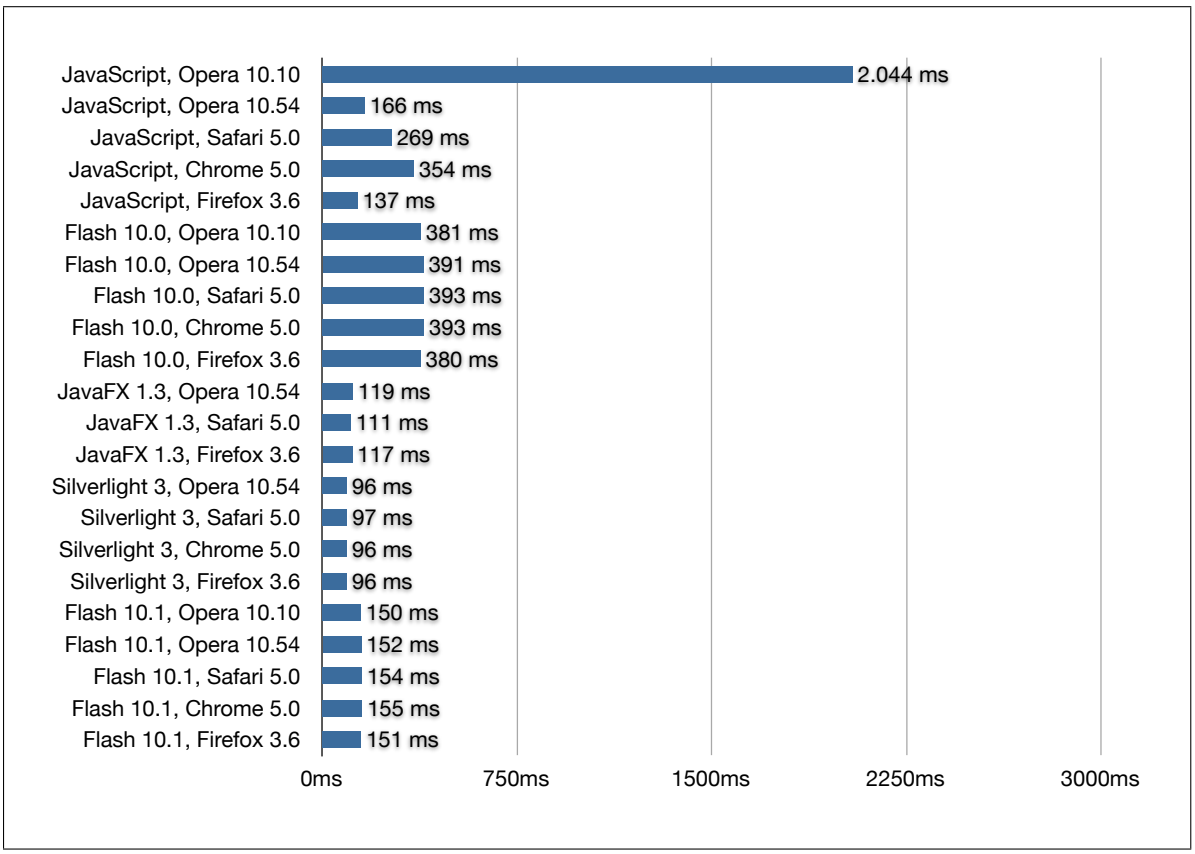

Figure 2.36: Results of the »MathTestCollection« on Mac OS X 10.6

nor differences of a few seconds. The only test where things change is the so called »PowerOfTestAPI«, where an existing method calculates x^n for given parameters x and n. Obviously, the implementation of this method for the Firefox-version is so good that the result of this test has a huge impact on the final numbers (see figure 2.37 on page 101). The non-API version of the test where this operation was emulated through the (*)-operator was not affected by this anomaly, which can be supported by the results of the »MathMultiplicationTest«. Really bad, but not unexpected, is the result form Opera 10.10 with 2044ms, which was the longest time elapsed in this test. Lastly to mention is that version 10.0 of Adobe's Flash was about 2-3 times slower (depending on the used browser) than the new player 10.1.

Regarding the results on **Microsoft Windows Vista**, it can be said that obviously almost all RIA runtimes returned very similar results (see figure 2.38 on page 102) except for Opera 10.10 and Internet Explorer 8.0 with again extremely bad numbers. As seen in the prior test-collection on Mac OS, again Firefox performed best on JavaScript (157ms) while Google's Chrome browser needed more than twice as much time to pass the test. The reason for this is that the browsers Chrome, Safari and Opera 10.54 performed extremely well on the tests for multiplication, addition, division, subtraction, modulo- as well as square-root calculation. But again, Firefox performed so well on

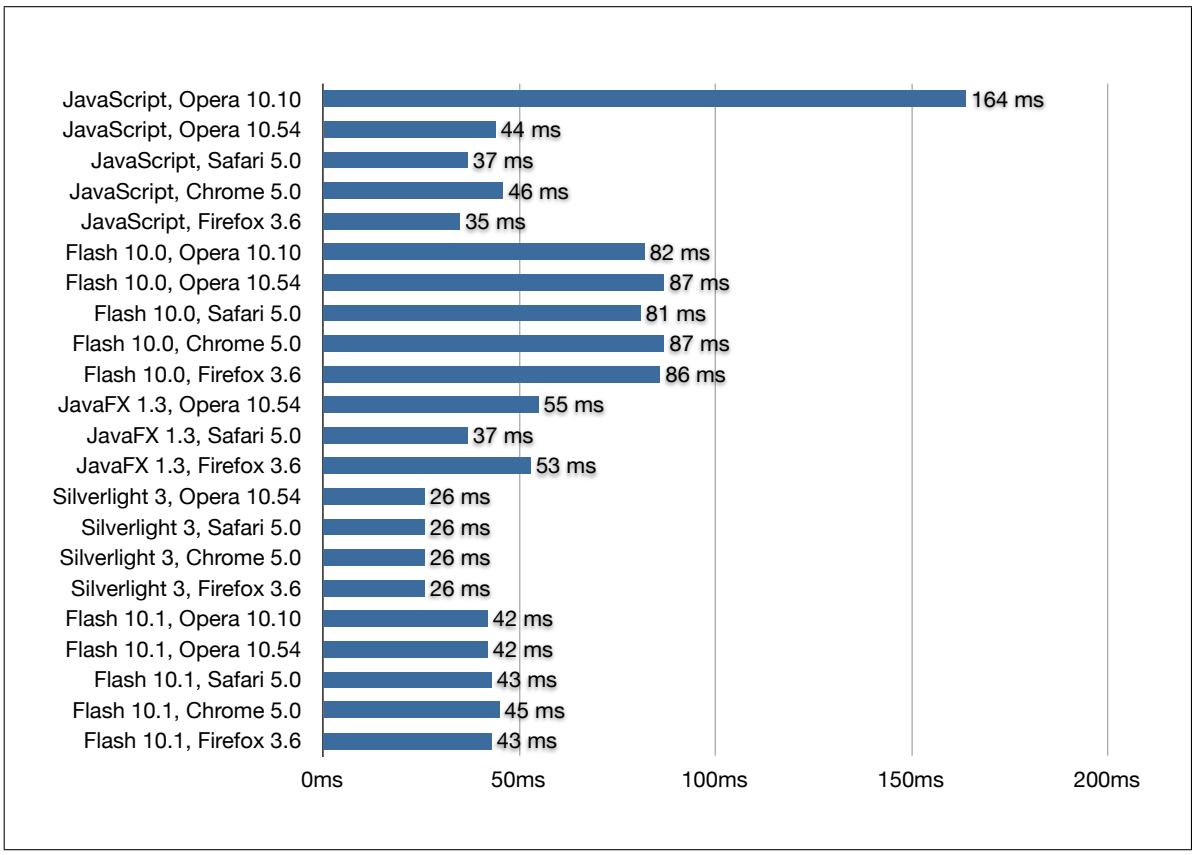

Figure 2.37: Result of the »MathPowerOfAPI« focus-test on Mac OS X 10.6

the »PowerOfTestAPI«, that the result of this single test had a huge impact on the final numbers of this test-collection. The most interesting result in Windows though was the difference between the results of the non-API version of the »PowerOfTest« and the multiplication-test. Since, test one is based on the (*)-operator, there should be no reason why the results should differ (see listing 2.3).

```
1  function powerOf(val:int, pow:int):int {
2   if (pow == 0)
3    return 1;
4   var res:int = val;
5
6   for (var i:int = 0; i < pow - 1; i++) {
7    res = res * val;
8   }
9   return res;
10 }
```

Listing 2.3: Own implementation of the Math.power API-function using the *-operator

In fact, it could be observed that Google Chrome for example returned the best results regarding plain multiplication but in the same time, it failed to perform well on the

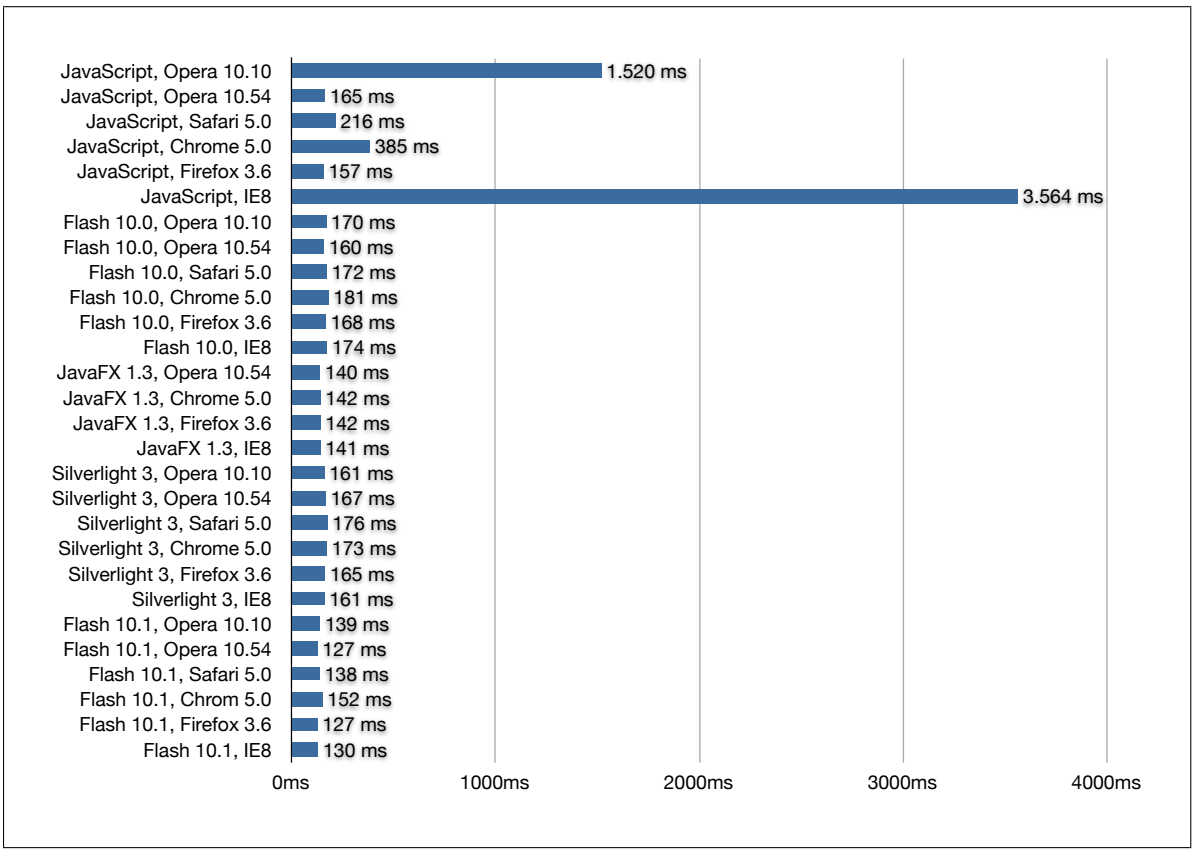

Figure 2.38: Results of the »MathTestCollection« on Microsoft Windows Vista

»PowerOfTest«. Currently, no explanation for this can be given at this point since further investigation onto this issue is necessary.

On **Ubuntu Linux 10.04**, the results are a bit more unbalanced (see figure 2.39 on page 103). Obviously Moonlight for Firefox 3.6 returned the best result at 90ms, followed by JavaFX on Firefox 3.6 (154ms) and Google's Chrome browser (171ms). The only runtime from the rest of the field that was able to catch was again Mozilla's Firefox 3.6 on JavaScript at 128ms. The other technologies are almost even between 193ms (Flash 10.1 on Firefox) and 379ms (Chrome 5.0 on JavaScript) except for Opera 10.10 which again returned with the worst values (by far) at 2588ms.

In summary, it can be said that both, Silverlight and JavaFX can be considered as the best runtimes regarding mathematic calculations across all operating systems. These observations can be supported by results from prior use-case-tests: While the prime-generation benchmark from section 2.3.2.1 on page 56 showed the opposite results (JavaFX being extremely slow) the prime-factorization benchmark proved the opposite. In order to understand how this could happen, the results of the »ArrayTestCollection« are necessary. These test-series showed that the insert-operation of arrays in JavaFX are absolutely inefficient. Since the algorithm used in the prime-test requires the storage of

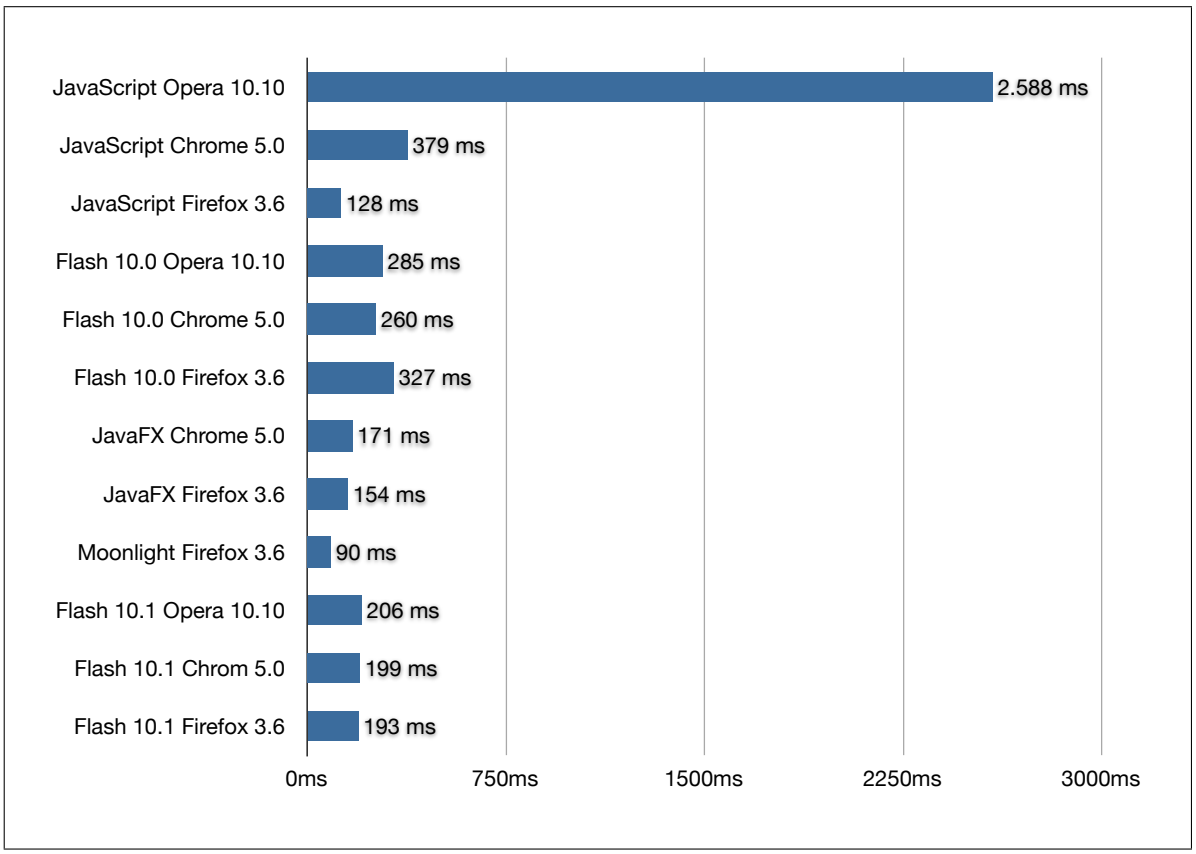

Figure 2.39: Results of the »MathTestCollection« on Ubuntu Linux 10.04

generated prime numbers into an array, it can be said that this is the cause for the great performance loss for JavaFX. Looking back at the results of the prime-factorization test (see section 2.3.2.2 on page 59), it could be observed that Firefox 3.6 already performed very well on that benchmark and hence it was assumed that mathematical calculations work well on this browser. This assumption could now be backed up by the results of this Focus-test.

3 Jaction

3.1 Concept

Jaction is a new, self-developed technique as a result of the prior use-case tests in this thesis. It is based on an own idea of using JavaScript technology in order to boost Flash application performance. The word has its origin in the two terms »**J**avaScript« and »**Action**Script« since the idea itself is a combination of both programming languages. As seen in the previous chapter, JavaScript often performs a lot better than Flash, especially on Webkit-based browsers like Apple's Safari or Google Chrome if compared to the Flash 10.0 (or lower) runtime. Even the new version 10.1 sometimes runs slower than certain JavaScript algorithms, depending on which operations are being used. Thus, two questions came up:

1. Is it possible to call JavaScript functions from within Flash to increase application performance?

2. Is this worth the effort?

3.2 Setup

Question number one can be answered easily: Yes it is possible, since the Flash plug-in already offers this functionality in the `flash.external` package (See fig. 3.1 on page 106). So, all that's needed to do is to use the `ExternalInterface.call(...)`-method as shown in listing 5.12 on page 132 (Using md5 hashing as an example here). The JavaScript implementation can either be put into a .js file or embed into the wrapping HTML markup. For the purpose of this demo, the second alternative will be used (see listing 5.9 on page 130), but both versions work fine. This is all that's needed to run this demo shown in the listing. If opened now in a web-browser with installed Flash plug-in, the Flex application will call the JavaScript-MD5 function, wait for it to complete and output the result to an alert box. The good thing about this method is its transparency: The JavaScript call is synchronous, which means that the programmer won't have to concern about any typical asynchronous issues like listening to completion-events,

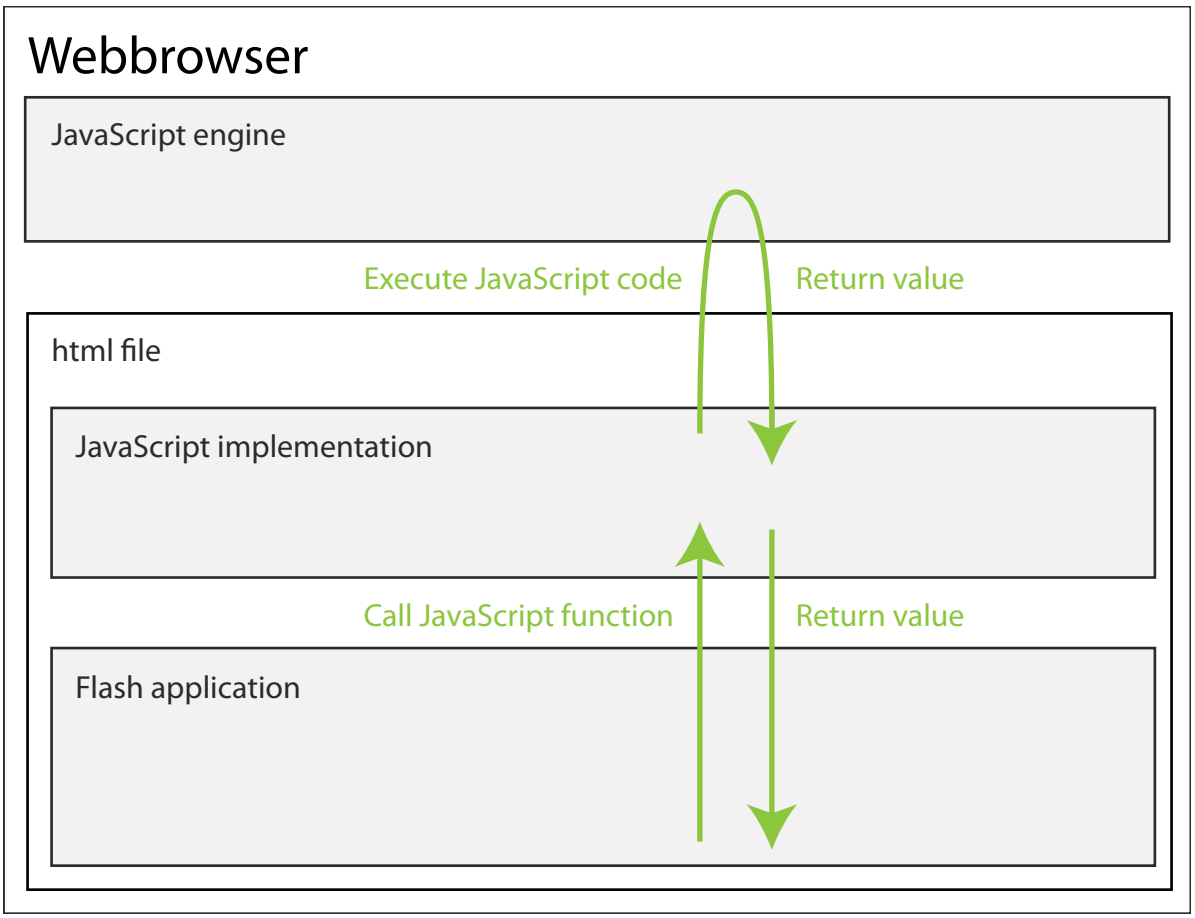

Figure 3.1: Calling JavaScript functions from inside Flash applications

which are often required in SOA architectures for example when calling web-services. At this point one could ask: »What is so great about the fact that Jaction-calls are synchronous?«, since the rise of the so called »Web 2.0« made asynchronous method calls very popular, especially due to excessive usage of Ajax[1] in many web-applications these days. The reason is that there are always two sides of a medal regarding asynchronous calls. One really good thing about these kind of invokes is that they cannot freeze an entire application because there is no direct waiting time for a method to return. On the other hand side, it is often not known when and if an asynchronous function call will return. The example in listing 5.10 on page 130 illustrates this dilemma. This short piece of pseudo-code simply calls a web-service, waits for it to return and then does something with its return-value. The good thing about this approach is that the programmer does not have to register any event-listeners. Instead, he can trade the RPC[2] like a local method invocation. The downside of this technique is that such calls over

[1] **A**synchronous **J**avaScripting **A**nd **X**ML: A technique to dynamically request and achieve XML data from within websites without re-loading the full page.

[2] **R**emote **M**ethod **C**all: A technique of invoking functions over distributed networks, like Local Area Networks or the Internet.

distributed networks sometimes tend to be either slow (due to network overload) or are completely unavailable (due to server-crashes or other network problems). In this case, the whole application would freeze until it either gets terminated by force or a timeout exception occurs (if the runtime environment is clever enough). However, both solutions are not optimal. The code snipped in listing 5.11 on page 131 does the same again, but based on an asynchronous approach. This version of the prior (synchronous) example uses the webservice in an asynchronous way, which means that the runtime will call the method `asyncWebservice.multiply(a, b)` but not wait for the service to return a value. Instead, the event-listener, which was added to the service, will get called once the asynchronous function returns. This approach provides a better user-experience since the program can continue to run instead of freezing the whole application. The downside though is that the method-call-chain gets interrupted, which means that the event-listener `success(result)` does not belong to the call-stack. Thus, it cannot return a value, hence the invocation of the function `doSomething(result)` will probably fail because the variable `result` does not contain a valid value. In order to make this code work, the call for `doSomething(result)` must be moved from the method `start()` to `success(result)`, like this:

```
function success(result):void{
 doSomething(result);
}
```

Now, the whole application works fine. However, as useful as such asynchronous approaches are in the context of remote procedure calls over distributed networks, like the World Wide Web, synchronous calls are the weapon of choice for Jaction. There are two reasons why:

1. Jaction was designed to boost performance of existing Flash-applications. Thus, it must be expected that Jaction-calls must be integrated into existing code. If method invocations expect a return-value of a Jaction-call, asynchronous implementations can lead to excessive code refactoring since all the lines of code, which require this return-value must be moved into the service's event-listener, as shown above.

2. Asynchronous method calls are usually only useful if the programmer must assume with the risk of having to wait a long time for the function to return, for example due to long network roundtrip-times or unavailability of remote services. This issue does not apply to Jaction-calls though, which are not invoked over a distributed system. Instead, the JavaScript-call is being done locally on the same computer. Thus, it is not expected that these invocations are delayed in any way.

Now, since Jaction is based on synchronous principles, this technique can be used like a native ActionScript call in Flash. There is nothing really to watch out for as long as the JavaScript function works fine. To guarantee this behavior and maximize the

robustness of the code in general, a try...catch block plus a verification of the result with a fallback-ActionScript method should make this code bulletproof (see listing 5.13 on page 132).

With this modification, the application can be stated as pretty robust since it can react on JavaScript errors and fall back to the original ActionScript implementation if something goes wrong. This is quite important since there exist many different JavaScript engines across various web-browsers. There is no guarantee that an implementation will work in browser A, only cause it did in browser B. In order to avoid these cross-browser issues, which might lead to slowdowns, a browser verification by checking their user-agent completes the Jaction prototype[3]:

```
1 // Assuming such a function exists
2 if (isJactionCompatibleBrowser()){
3  hash = hashWithJaction(strToHash); // Do the Jaction call
4 }
5 else{
6  hash = MD5.hash(strToHash); // Do the normal AS3 call
7 }
```

Listing 3.1: Verification if the used browser is Jaction-compatible

Of course, the method `isJactionCompatibleBrowser()` does not exist in any JavaScript API in the world because Jaction is a new, own invention made during the process or writing this thesis. Thus, in order to provide a method if Jaction-calls are possible in certain browsers and operating systems, the Jaction-Framework was developed. See chapter 3.6 on page 116 for more information.

After the creation of this prototype was finished, it must be said that using Jaction does definitely leads to more effort regarding coding. Since it can happen that required algorithms must be implemented twice (worst case) and various fall-back techniques must be used (in order to guarantee stability), the question now must be if this additional work is worth the potential performance increase.

3.3 Is using Jaction worth the effort?

This question is a little more difficult to answer. Thus, some experiments were needed. The browser of choice for this is Google Chrome, since Chrome has one of the fastest JavaScript engines according to tests from sixrevisions.com[Gub] (see figure 3.2 on page 110). Test one is a simple iteration over the variable i from 0 to 700 where sqrt(i) should be calculated on each iteration step. This was first implemented in two different ways using a for-loop:

[3]For the complete source of the prototype shown here, see the attachments on page 133.

- **Jaction test 1:** Pure ActionScript:

```
for (var i:int=0; i<=700; i++){
 Math.sqrt(i);
}
```

- **Jaction test 2:** Only the for-loop is coded in ActionScript. The sqrt(i)-call was replaced with its JavaScript pendant:

```
for (var i:int=0; i<=700; i++){
 flash.external.ExternalInterface.call("Math.sqrt", i);
}
```

The result was astonishing and absolutely not expected: Test one completed in less than 1ms while number two terminated after 7571ms. The cause for this could only be one of the following: Either the `ExternalInterface.call`-method takes very long to call the JavaScript function or the `Math.sqrt`-implementation itself was to slow. In order to verify what the reason for this huge performance loss was, another variant was added to the test: This time, the complete for-loop was swapped out to JavaScript, which leads to only one `ExternalInterface`-call:

- **Jaction test 3** (ActionScript):

```
flash.external.ExternalInterface.call("mySqrt2", 0, 700);
```

The JavaScript implementation in the wrapping HTML template looks as one would expect it:

- JavaScript implementation for Jaction test 3:

```
function mySqrt2(start, end){
 var res = new Array();
 for (var i=start; i<=end; i++){
  res.push(Math.sqrt(i));
 }
 return res;
}
```

This time, the elapsed time was 2ms. which was a bit more than the pure ActionScript version needed but still way less than Jaction test 2. The minimal higher number of milliseconds is probably because of the `Array.push(...)`-operation in each iteration which is required to return all results at once and can thus be ignored.

This result can lead to only one conclusion: Calling JavaScript functions from within Flash is expensive and takes quite long, depending on the machine the application runs on. Based on this insight, it can be said that the number of JavaScript calls must be minimized. More precisely: The less the number of JavaScript calls, the bigger the chance to benefit from using Jaction.

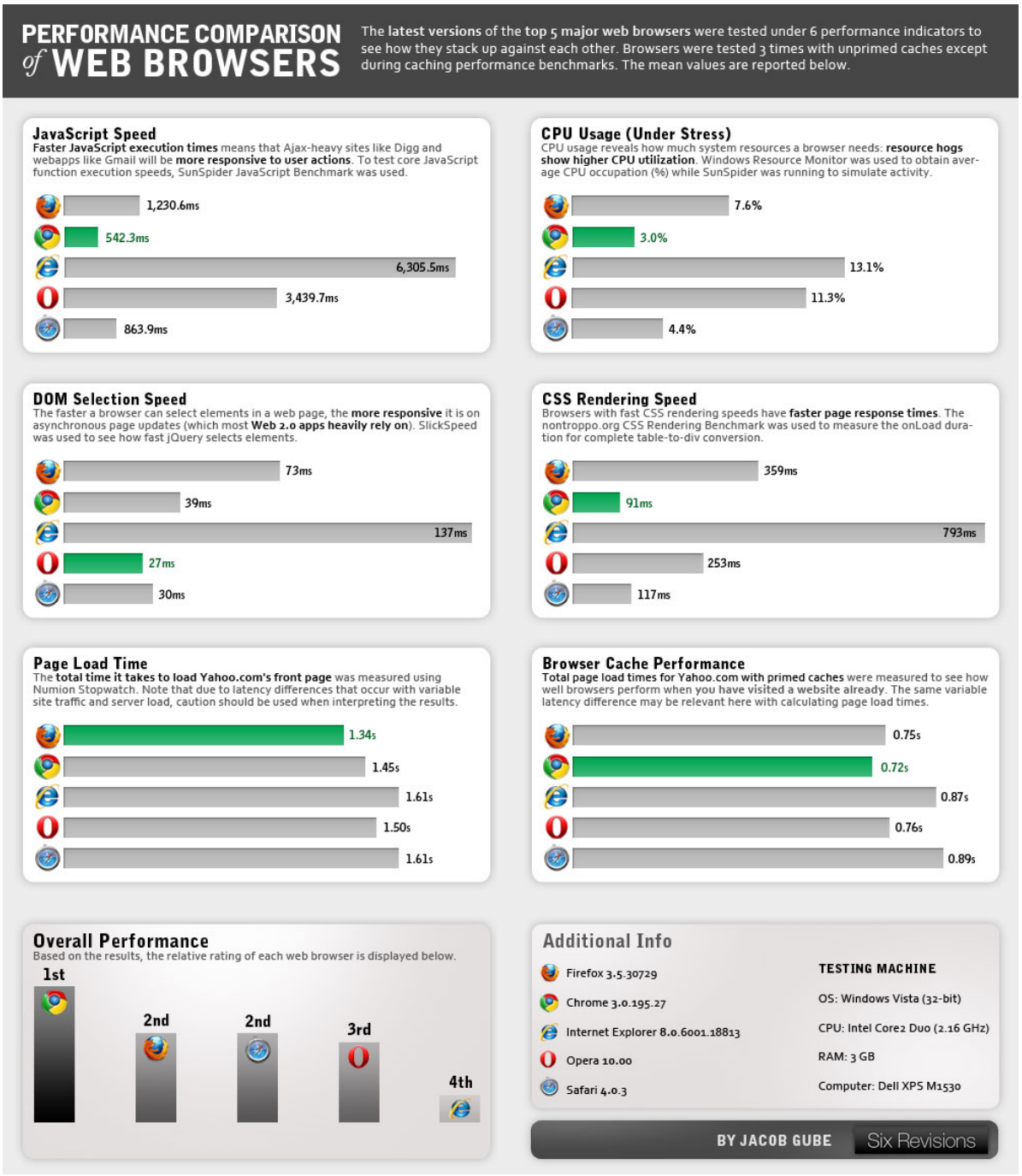

Figure 3.2: Webbrowser-speed evalution from sixrevisions.com[Gub] by Jacob Gube

3.4 Demo: JPEG encoding using Jaction

Jaction test number three showed that there was no performance increase, although the number of JavaScript calls was equal to one. The reason for this could be that Math.sqrt() is not complex enough to make use of the fast JavaScript engine of Google Chrome. To examine this issue, another test for compressing images using the JPEG algorithm has been setup.

3.4.1 Test setup

In order to test JPEG compression once with pure ActionScript and once using Jaction, a Flex project was set up. The application consists of a drop-down menu, which lets the user choose between one of the two techniques. On selection, a given PNG image (Same as in section 2.3.1.1. See fig. 2.3 on page 36) will be compressed using the JPEG algorithm provided in the Flex 3.2 SDK (ActionScript version) and by Andreas Ritter[Rit10] (JavaScript version). See fig. 3.3 on page 112 for a screenshot of the demo program from `http://www.timo-ernst.net/jaction`. On encoding start, a timer will record the elapsed time. When the compression is done, the timer will be stopped and the result displayed to the right of the drop-down menu. In order to be able to see if errors occurred, the fall back technique using a try...catch block, as introduced in section 3.2 on page 105, will not be used. The test was done on MacOS X (Snow Leopard) on a Macbook Pro with a Intel Core2Duo running at 2,53GHz using 8 GB RAM.

3.4.2 Why JPEG?

There are three reasons why JPEG encoding is a good test for this purpose:

1. The JPEG encoding algorithm is rather complex than simple. It takes some seconds to compress an 1024x768 image, which is just right for the purpose of this test.

2. As stated in section 2.3.1.1 on page 36, the JavaScript implementation by Andreas Ritter[Rit10] is based on the original ActionScript-version made by Adobe. Thus, the test results won't be influenced to strong by different implementations. The differences regarding syntax between both scripting languages are minimal while the algorithm implementation itself should be almost equal. This means that differences regarding the resulting computing times between pure ActionScript and Jaction-versions will be rather caused by the different scripting engines, than by the implementation of the compression algorithm.

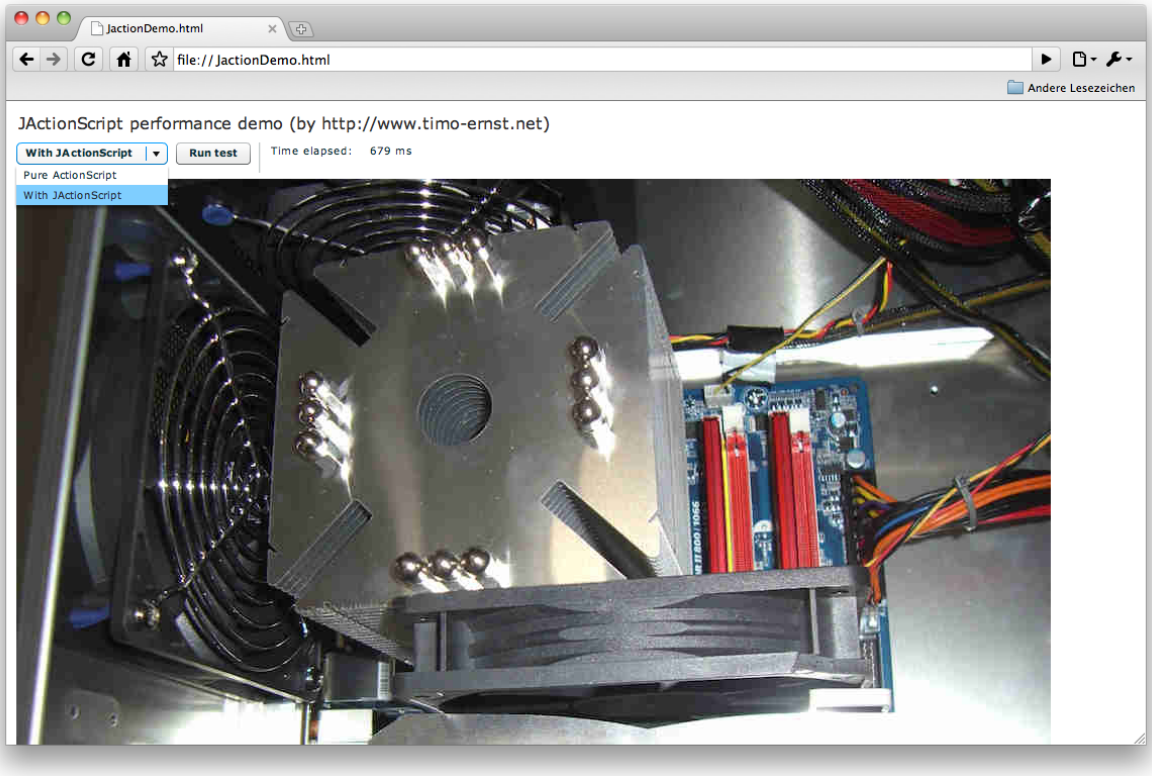

Figure 3.3: Jaction demo (JPEG compression)

3. Since the resulting, compressed image will be displayed on the screen after the test, it will be easy to see if the compression algorithm worked or not because JPEG encoding is not a loss-less compression method, as stated in section 2.3.1.1. Using settings for rather bad image quality should make artifacts visible (See fig. 2.4 on page 37).

3.5 Result

The result can be split into three groups: Browsers, where Jaction leads to a performance increase, which would be Google Chrome and Safari, those where a slowdown is the consequence (Firefox) and the ones where Jaction doesn't work at all (Opera 10.10 and 10.54). The following table shows how much faster Jaction was compared to Flash 10.0 and 10.1:

Browser	**O.S.**	**Accel. vs. Flash 10.0**	**Accel. vs. Flash 10.1**
Chrome 5.0	Mac OS X	116%	52%
Safari 5.0	Mac OS X	154%	78%
Firefox 3.6	Mac OS X	-39%	-83%
Opera 10.10	Mac OS X	incompatible	incompatible
Opera 10.54	Mac OS X	incompatible	incompatible
Chrome 5.0	Ubuntu 10.04	200%	119%
Firefox 3.6	Ubuntu 10.04	-74%	-121%
Opera 10.10	Ubuntu 10.04	incompatible	incompatible
Chrome 5.0	Windows Vista	75%	20%
Safari 5.0	Windows Vista	48%	8%
Firefox 3.6	Windows Vista	-76%	-149%
Opera 10.10	Windows Vista	incompatible	incompatible
Opera 10.54	Windows Vista	incompatible	incompatible
Internet Ex. 8.0	Windows Vista	incompatible	incompatible

The results can basically be separated into three groups:

- Browsers where Jaction leads to performance acceleration:
  - Chrome 5.0 across all operating systems
  - Safari 5.0 across all operating systems
- Browsers where Jaction leads to performance loss:
  - Firefox across all operating systems
- Browsers where the Jaction demo did not work:
  - Opera in both versions across all operating systems
  - Internet Explorer 8.0 in Microsoft Windows

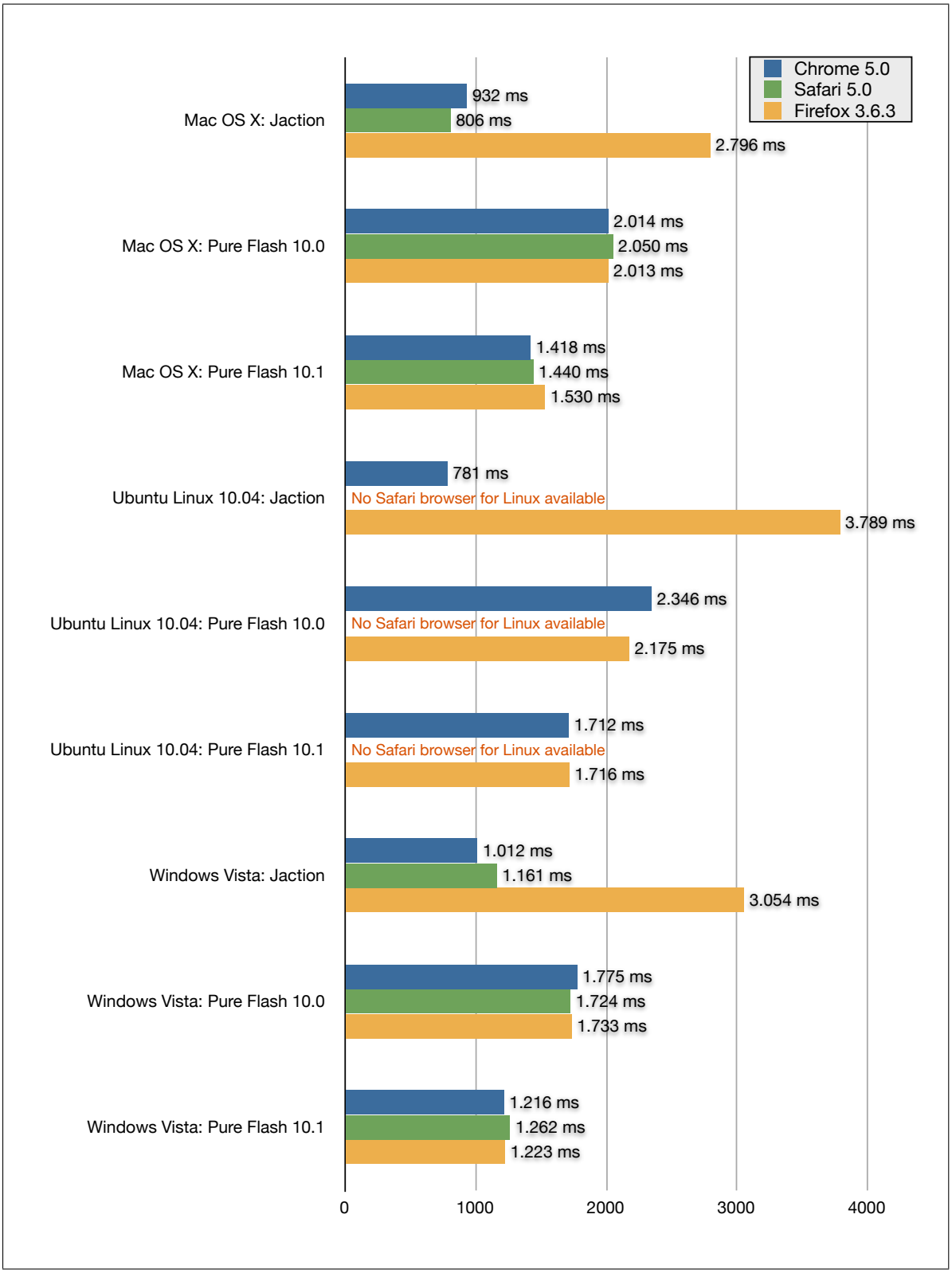

Figure 3.4: Results of the JPEG-encoding test using Jaction

The best acceleration rates could be produced on Google Chrome for Ubuntu (200% acceleration), and Safari 5.0 on Mac OS X (154%), both against Flash Player 10.0. Obviously, the new version 10.1 of Adobe's runtime is about two times faster than its predecessor in most cases but even a small amount of speed increase could be observed on the lowest acceleration result for Safari 5.0 on Windows Vista (8%). Hence, it can be said that Jaction is always worth using on Chrome as well as Safari as long as all other requirements for this technique (e.g. »minimum number of Jaction-call, etc...) are met. Regarding the Firefox 3.6 browser, it can be said that Jaction should not be used because a performance de-acceleration is actually very possible on all operating systems (see table above). On Windows Vista, the Jaction version was even 149% slower compared to Flash Player 10.0. No results could be made on Internet Explorer 8.0 due to the fact that it lacks support for the `<canvas>` element, which is absolutely necessary for this test, as described in section 2.3.1.1. Regarding the results for Opera, it must be said that Jaction is currently not possible on this browser because all attempts to call the outer JavaScript-functions from within Flash did not succeed. — Actually the call itself worked but no return value could be achieved. It is assumed that the reason for this are JavaScript interrupts caused by Opera itself in order to refresh the visual DOM representation, since this behavior could already be observed in previous stand-alone tests (See section 2.3.1.1 for instance) and does make sense in order to not let the browser freeze due to very long running JavaScript tasks. While stand-alone tests are no problem if called for themselves, the Jaction version simply terminates after 101ms without doing anything due to these interrupts.

Interpretation

Based on the results of the tests, it can be said, that Jaction does not always lead to a performance increase. The answer to the question whether one should use this technique or not is connected to three important aspects:

1. As seen in section 3.3 on page 108, the algorithm which needs to be implemented in JavaScript, must be complex enough. If not, the performance increase will be minimal or even negative, which leads to a performance decrease instead. This requirement of being »complex enough« was not investigated in a more detailed way. For the moment, it must be said that developers simply have to try if their algorithm works using Jaction or not.

2. In the same section, it was observed that the number of JavaScript calls must be minimal, since the `ExternalInterface.call()` method takes quite a while to do its job. Currently, there is no news in the press that Adobe will improve this feature in the next time, so for the moment, this issue can be considered as a bottleneck if not explicitly handled.

3. Jaction does not lead to a performance increase on all browsers. Currently only browsers that use the Webkit engine can benefit from this technique, since other JavaScript implementations are currently too slow or simply refuse to delegate the calls from within Flash at all. While Firefox does not benefit from Jaction, Opera is incompatible with this technique in general.

3.6 The Jaction-framework

As mentioned in the sub-sections before, Jaction does not always work. More precisely: The technique only leads to performance improvements on certain combinations of web-browsers, operating systems and Flash Player versions. Thus, during the creation of this thesis, the so called Jaction-Framework was developed which can be used in order to verify wether a Jaction-call does make sense or not. The usage of the framework is rather simple. See the code in listing 3.2 on page 116 for a quick example.

```
1  import de.timoernst.jaction.Jaction;
2  private function init():void {
3   if (Jaction.isJactionCompatible()) {
4    var param1:int = 1;
5    var param2:int = 2;
6    Jaction.call("myJavaScriptFunc", param1, param2);
7   }
8   else {
9    // Do a native, non-Jaction-call
10  }
11 }
```

Listing 3.2: Example usage of the Jaction framework

The method `isJactionCompatible()` returns true if the browser, as well the Flash Player used, match the requirements in order to let Jaction increase the performance of Flash-applications. While the Flash Player version can easily accessed directly from within Flash applications, browser specifications are more difficult to achieve. In order to implement this, some JavaScript code was required, which must be put inside the HTML-file, which wraps the SWF containing the Flash application. The function `call(jsFunction:String, ... arguments)` does a Jaction-call. `jsFunction` is the name of the JavaScript method, which should be called. All parameters after this argument can be used to pass additional data to this function (must be separated by commas).

The full Jaction-Framework source-code can be downloaded from:
`http://www.timo-ernst.net/jaction/framework/jaction_framework.zip`

4 Conclusion

»Whenever anyone says, ›theoretically‹, they really mean ›not really‹«

(Dave Parnas, early pioneer of software engineering and developer of the concept of information hiding in modular programming)

4.1 Performance analysis

Theoretically, there should be no reason why a plugin-based technologies, like Adobe Flash or Sun's JavaFX platform, should be influenced by the browser (which »wraps« applications created with one of these techniques) regarding running performance since these »add-on's« are external runtimes, which should thus not interfere with the browser's execution- or rendering-engine. In this thesis it could be shown that there are a few cases where this can actually happen. The following paragraphs sum up these insights and give more detailed information when and why these »anomalies« occurred together with the most interesting results among all series of performance tests.

Opera 10.10 and Internet Explorer 8.0 almost always returned by far the worst **JavaScript**-performance. Microsoft's browser, for example, performed 629 times slower than the best runtime (JavaFX, Opera 10.54) on Windows Vista in the MD5-test (see section 2.3.1.2 on page 40). Version 10.54 performed very well on JavaScript on all tests and was sometimes even faster than the top Webkit-based browsers Google Chrome 5.0 and Safari 5.0. Therefore, it can be recommended for users of Opera 10.10 to upgrade to the new version since performance boosts of factor 10 (and more, depending on the use-case) are possible (see figure 4.1 on page 119). Firefox on JavaScript performed in average of the whole field, except for the »MathPowerOfTest(API)« where Mozilla's browser returned excellent results.

Using the (+)-operator for string-concatenation does not lead to performance decreases. Using a string-buffer, based on arrays, fails due to slow operations on these but if a lot of textual data must be displayed to the user, it should be avoided to update the components, which contain these strings, in very short intervals because it could be shown that this kind of »string-dumping« can lead to heavy loss of performance. Instead,

one should try to create larger »chunks« of textual data, store them into a buffer and display its content to the user at lower intervals.

Regarding 3D-performance, no conclusion could be made due to the fact that WebGL, a very promising 3D-API for JavaScript-based applications, is only supported in nightly builds of the browsers tested in this thesis. Since these versions are often highly experimental and unstable, it was decided to not implement a 3D-test for JavaScript.

Another interesting thing to mention is that, although both browsers, Google Chrome and Safari from Apple, share the same engine called »Webkit«, their results do not always match. The MD5-test in the series of usecase-tests for example, showed that Safari was twice as slow than compared to its competitor from Google.

Furthermore, regarding HTML5's new `<canvas>` element, it can be said that this new technology cannot compete with other solutions yet and is still inferior to e.g. Flash 2D-performance as shown in section 2.3.2.5.

Lastly, it could be observed that memory leaks have a much heavier impact on JavaScript-based applications than on all other technologies. The benchmark in section 2.3.2.6 showed that performance can easily go down the more objects are being created. Thus, it can only be recommended to explicitly destroy unused objects by using the `delete` operator instead of relying on the browser's own garbage collection.

JavaFX did very well across all operating systems most of the time, but there are some drawbacks as well. One is that operations on arrays (also called »sequences« in JavaFX-terms), seem to be implemented absolutely inefficiently (See figure 2.33 on page 96 for example). Especially insert-operations are way slower than compared to the performance of competitors.

Furthermore, it could be observed that string-concatenation using literals (through the {}-operator) is extremely slow if many iterations must be processed. Therefore, based on the insights in this thesis, it is recommended to use the StringBuffer class. The fact that JavaFX performs so slowly on »ordinary« string-concatenations did probably lead to the bad results in the Run-length-encoder test, which heavily relies on this. Other string-operations like searching and splitting worked fine.

The third downside of JavaFX was the lack of support for plug-ins in some browsers like Opera 10.10 (all operating systems), Google Chrome 5.0 (Mac OS X only) and Safari 5.0 (Microsoft Windows). This drawback can be considered as a huge problem since users cannot benefit from the fastest runtime-environment if it is simply not available to them. Problems running the 3D-test on Mac OS X, caused by an Apple-modified version of the Java runtime, made it impossible to run this benchmark although the 3D-performance on Windows and Linux were the best among all RIA technologies.

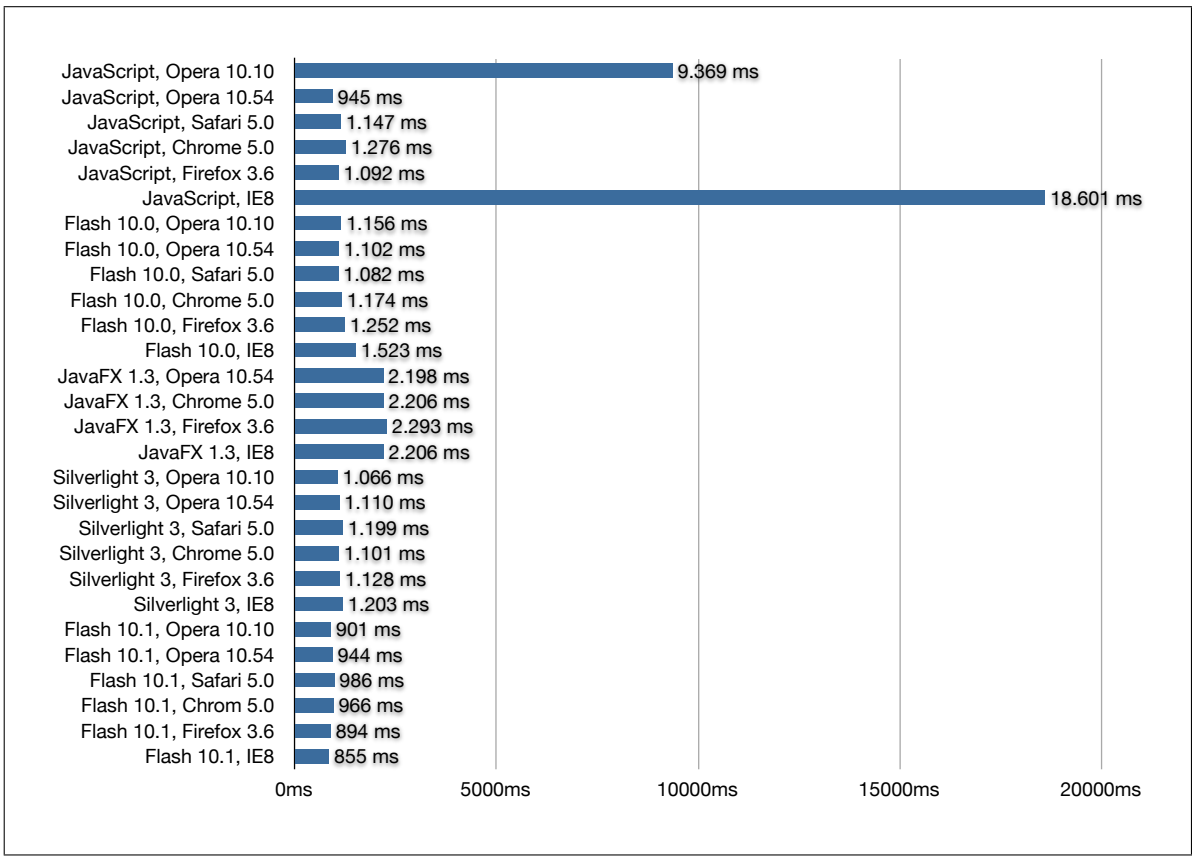

Figure 4.1: Final results of the »Focus-test« on Microsoft Windows Vista

Furthermore, a minor but not unimportant thing to mentioned is that JavaFX applications tend to load very long if deployed to a browser (similar to classic Java applets). Where Flash- or Silverlight-applications only needed seconds to load, JavaFX required much more time until the program was ready to use. This observation was not a result of a specific test but it was definitely noticeable during the development and testing of the benchmarks.

Lastly, heavy performance losses were noticed on JavaFX on Google Chrome on the 3D-test while the same test with the same plugin on other browsers ran way faster (see figure 2.11 on page 54). This can be considered as proof that there is a chance that browsers can influence the performance of plugin-based applications. This hypothesis can also be backed up by the results of the prime-generation test, described on page 56. On Mac OS X, the prime-test for JavaFX finished in only 288ms while the other versions with the same runtime but on different browsers, needed much more time to finish the test (1454-1787ms). Obviously, browsers can not only negatively but also positively influence a plugin-based technology's runtime performance.

Regarding **Silverlight**, it must be said that both versions from Microsoft as well as the

one from Mono, called Moonlight, performed very well on most benchmarks. Moonlight did excellent in most cases except for those tests where the original Silverlight plug-in did perform bad anyway (e.g. 3D-test) but in many cases, Mono (version 3, beta) was actually faster than the one from Microsoft. The only two tests where both failed are the 2D- and 3D tests. Currently, it is unknown if the used framework Kit3D (in the 3D-test) was the cause for these extremely low frame-rates or a problem with the runtime itself lead to these. Since both tests were not hardware-accelerated and Silverlight usually performed very well on all other tests, it can be assumed that the framework was the cause since the API-function provided were very limited and thus inefficient anyway. Regarding the results of the 2D-test, it is absolutely unexplainable why Silverlight performed so badly. At the moment, the only explanation can be a slow 2D-drawing engine or an inefficient implementation of the test.

Looking at string-concatenation operations, it could be observed that it is very important to use the StringBuffer-classes, as described in the prior paragraph for JavaFX since the (+)-operator as well as the `str.concat(str2)`-methods are absolutely inefficient.

About **Flash** runtime performance, it can be said that using the (+)-operator for string-concatenation does not lead to speed decreases. Using a string-buffer, based on arrays, works but does not increase performance significantly. Regarding the two tested versions of Flash, only on Mac OS X, a noticeable boost by about 200% could be observed between version 10.0 and 10.1. These observations can be supported by results from the 3D-test where an increased frame-rate was the consequence of an upgrade from 10.0 to 10.1. On Windows almost no increases could be observed while the Linux version at least benefited from the new version by about 50%. 3D-performance in Flash was neither bad nor overwhelming (compared to some results on JavaFX). Currently it is unknown if the framework »Papervision« was the cause or the player runtime itself. Due to the fact that Flash does not support hardware-accelerated 3D-engines, the 2nd assumption is more possible though. For a completely in software implemented and rendered application, the 3D-performance in Flash was tolerable though since most results showed fps rates around 21 fps on version 10.1, which is just enough so that human beings don't perceive animations as being to »laggy«. The older version 10.0 did a bit worse though, especially on Mac OS X.

A last but important observation which could be made on this 3D-test on Flash running in the Opera 10.10 browser, where a significant loss in fps could be seen, while the same runtime did quite well on other browsers on the same operating system (Mac OS X). Although the cause for this issue was not investigated any further, it is assumed that Opera's rendering engine seems to interfer with Flash's own implementation. However, this fact can be considered as proof that obviously Opera did influence Flash Player's performance in this test (see figure 2.12 on page 55 for a chart visualizing this issue).

Regarding the **letter of Steve Jobs** and his accusations about Flash being slow and unstable, especially on mobile devices, it could not be proven that these are correct. On

Mac OS, the new Flash Player 10.1 brought a lot of improvement regarding performance while on Windows and Linux, almost nothing changed. Since it is very likely that Apple's CEO uses Macintosh computers rather than Windows or Linux, it is possible that Mr. Jobs experienced the performance of version 10.0, which indeed is pretty slow. 10.1 though brings a huge boost to Apple computers and is thus a »must-install«. Regarding crashes caused by the Flash Player: During all tests, no stability issues were noticed.

Tests on mobile devices (like for example Android[1]-based smartphones) could not be made, since Flash Player (10.1) is only available on Android 2.2 devices and the author's smartphone (HTC Desire) does not yet support it. Thus, Job's accusations about Flash being an inappropriate technology for mobile devices could not be examined.

In summary it can be said, that there is no »perfect« runtime, which can be used in any project, guaranteeing no performance-problems. Thus, it is important to know the strengths and weaknesses of existing technologies and combine this knowledge together with the requirements of these projects. Among all benchmarks it could also be observed that the API-versions were usually a lot faster than the Non-API variants. Thus, re-writing existing implementations does not make sense in most cases. Regarding the measurement of performance it can be said that it is not enough to run only one benchmark like GUIMark or Google's V8. In the same time it must also be said that the Focus-test, which was introduced in this thesis, alone is insufficient although it is very useful once suspicious behavior could be observed. It is important to combine multiple tests and »drill down« to the core of performance bottlenecks in order to find the real cause for e.g. frame-drops or slowly reacting user-interfaces.

4.2 Jaction

As seen in the previous sections, Jaction *can* lead to application speed benefits but there is no 100%-guarantee that it will. Sometimes, even performance decreases or the loss of functionality can be the result. There are various parameters which have to be kept in mind if one decides to use this technique or not, which are:

1. Type and version of the operating system
2. Flash-player type and -version
3. Browser-type and -version
4. Number of external JavaScript-calls

[1] An open-source operating system based on Linux, created by the Google corp.

5. Runtime-complexity or the delegated JavaScript-algorithm

While issues 1-3 can be delegated to the Jaction-framework, number 4 and 5 require some thoughts about the code, which should be executed through the external JavaScript-engine. One additional, interesting thing to mention about Jaction is that it pushes Flash applications one step further to the concept of open source technology. Since swf[2] files, which are used for deploying Flash applications, are in a semi-binary[3] form, their source is not visible. There is a compiler option to enable source-viewing in the current Flash SDK 3, but only a minority of the developers make use of this since it has to be explicitly enabled. Using Jaction, programmers who want to benefit from its performance boost, are being forced to show at least JavaScript source which is always visible to the user. Besides these facts, Jaction can also be a bridge towards a Flash-free web where JavaScript-based engines replace the Flash player. Although the Flex SDK and Builder are great tools for creating next-gen web applications, the Flash player often doesn't perform very well, as shown in the prior sections in this thesis.

Furthermore, it would be possible to create a framework for Flash applications which implements ActionScript functions in JavaScript by encapsulating those classes from the Flash/Flex API that would benefit from Jaction. Then, for example, it would become possible for developers to use `Jaction.mx.graphics.codec.JPEGEncoder` instead of `mx.graphics.codec.JPEGEncoder`. If implemented in a robust way, as shown as in section 3.2 on page 105, there is nothing that should go wrong. The Jaction-API could check which browser and operating system the user runs on and then decide whether the Jaction or ActionScript API should be used. All that's left to do for the developer is to import the framework to his project and add the String »`Jaction.`« to all `import` statements. He won't have to worry about anything else, since the framework itself takes care about the rest.

Finally, the concept of Jaction is not bound to the combination of Flash and JavaScript. Instead, the idea itself can be adapted to other technologies as long as three conditions are met:

1. There must be at least two runtime environments where one wraps the other.

2. The outer execution engine must be faster than the inner.

3. It must be possible to call outer methods from the inner environment.

If these requirements apply, the usage of Jaction is possible no matter on which combination of runtimes and execution engines.

[2]ShockWave Format
[3]Parts of SWF are actually »parseable«

Bibliography

[Ado] Adobe: *Rich Internet applications.* `http://www.adobe.com/resources/business/rich_internet_apps`. – Last date of visit: January 2010

[Bri] Brimelow, Lee: *SWF Framerate Optimization.* `http://www.gotoandlearn.com/play?id=112`. – Last date of visit: January 2010

[Chr] Christmann, Sean: *GUIMark.* `http://www.craftymind.com/guimark`. – Last date of visit: January 2010

[Daw] Dawson, Mark: *Kit3D.* `http://www.markdawson.org`. – Last date of visit: January 2010

[Duh03] Duhl, Joshua: *Rich Internet Applications (Whitepaper).* November 2003

[Gav] Gavrilov, Alexey: *Bubblemark benchmark.* `http://www.bubblemark.com`. – Last date of visit: January 2010

[Gro05] Grosso, William: *Laszlo: An Open Source Framework for Rich Internet Applications.* Version: March 2005. `http://today.java.net`. – Last date of visit: January 2010

[Gub] Gube, Jacob: *Performance Comparison of Major Web Browsers.* `http://sixrevisions.com/infographics/performance-comparison-of-major-web-browsers`. – Last date of visit: May 2010

[Imb10] Imbert, Thibault: *Optimizing performance for the Flash platform.* 2010

[Jun95] Jungblut, Ralf: *Kryptographische Zufallszahlengeneratoren (Whitepaper).* 1995

[Kar07] Kargl, Prof. Dr. F.: *Sicherheit in IT-Systemen (Lecture notes).* 2007

[Kir09] Kirkpatrick, Andrew: *Accessible Rich Internet Applications with Flash, Flex, and AIR (Whitepaper).* September 18, 2009

[Lam] LAMMERSDORF, August: *FXCanvas3D*. `http://www.interactivemesh.org`. – Last date of visit: January 2010

[Moo] MOOTOOLS: *Slickspeed - Speed/validity selectors test for frameworks*. `http://www.mootools.net/slickspeed`. – Last date of visit: January 2010

[Mor10] MORRIS, Simon: *JavaFX in Action*. 2010

[Rit10] RITTER, Andreas: *A JPEG Encoder for JavaScript*. Version: January 2010. `http://www.bytestrom.eu/blog/2009/1120a_jpeg_encoder_for_javascript`. – Last date of visit: January 2010

[Riv92] RIVEST, Ronald: *The MD5 Message-Digest Algorithm (RFC 1321)*. 1992

[Rog07] ROGOWSKI, Ron: *The Business Case For Rich Internet Applications (Whitepaper)*. March 2007

[Vel08] VELICHKOV, Peter: *Dojo vs JQuery vs Mootools vs Prototype performance comparison*. Version: February 2008. `http://blog.creonfx.com/javascript/dojo-vs-jquery-vs-mootools-vs-prototype-performance-comparison`. – Last date of visit: January 2010

[Web04] WEBER, Prof. Dr. M.: *Mediale Informatik (Lecture notes)*. 2004

[Web09] WEBER, Prof. Dr. M.: *Web Engineering (Lecture notes)*. 2009

[Wik] WIKIPEDIA: *Rich Internet Applications*. `http://en.wikipedia.org/wiki/Rich_internet_application`. – Last date of visit: January 2010

[WW05] WANG, Xiaoyun ; (WHITEPAPER), Hongbo Y.: *How to break MD5 and other hash functions*. 2005

5 Attachments

5.1 Source-code

5.1.1 Use-case tests

```
int x, y, z;
Transform3D transformer;
TransformGroup tGroup;
// (...)
// Add each planet to the TransformGroup
for (int i=0; i<numOfPlanets; i++){
 // Get new (pseudo-)random coordinates from the generator
 x = getNewCoordinate();
 y = getNewCoordinate();
 z = getNewCoordinate();

 // Position planet i in the solar system
 transformer = new Transform3D();
 transformer.setTranslation(new Vector3f(x, y, z));
 tGroup = new TransformGroup();
 tGroup.setTransform(transformer);
 tGroup.addChild(sphere);

 if (sunTransformGroup != null){
  /* sunTransformGroup is a global variable of type TransformGroup
     containing all planets which should rotate around the sun */
  sunTransformGroup.addChild(tGroup);
 }
}
(...)
```

Listing 5.1: 3D-rotation with Java3D

```java
/**
 * Will convert the given image (filename), compress it to JPEG
 * and return the result
 */
public class JpegEncoder {
 /**
  * Does the actual compression
  * @param inputFileName Path to the image to compress
  * @return The compressed image as BufferedImage object
  * @throws Exception
  */
 public BufferedImage encode(String inputFileName) throws Exception{
  InputStream inputStream;
  inputStream  = this.getClass().getResourceAsStream(inputFileName);
  BufferedImage renderedImage = ImageIO.read( inputStream );
  ByteArrayOutputStream tmpOutputStream = new ByteArrayOutputStream();
  Iterator iter = ImageIO.getImageWritersByFormatName("jpg");
  IIOImage iioi = new IIOImage( renderedImage, null, null );

  // Get the ImageWriter
  if (iter.hasNext()){
   ImageWriter writer = (ImageWriter) iter.next();
   ImageOutputStream ios;
   ios = ImageIO.createImageOutputStream( tmpOutputStream );
   writer.setOutput(ios);
   JPEGImageWriteParam iwparam = new JPEGImageWriteParam(Locale.US);
   iwparam.setCompressionMode( ImageWriteParam.MODE_EXPLICIT ) ;
   iwparam.setCompressionQuality( 0.2f );

   // Start the encoding process
   writer.write( null, iioi, iwparam );
   ios.flush();
   writer.dispose();
   ios.close();
  }

  // Write the result into the BufferedImage object and return it
  InputStream tmpInputStream;
  tmpInputStream = new ByteArrayInputStream(tmpOutputStream.toByteArray
     ());
  BufferedImage encodedImage = ImageIO.read(tmpInputStream);
  return encodedImage;
 }
}
```

Listing 5.2: Jpeg-encoder written in pure Java

```
// Example source code fragment for planets with "earth" textures
[Embed(source="earth.png")] private var EarthTexture:Class;
var earthTexture:Bitmap = new EarthTexture() as Bitmap;
var earthMaterial:BitmapMaterial
 = new BitmapMaterial(earthTexture.bitmapData);
var planet
 = new Sphere(earthMaterial, planetSize, planetSize/5, planetSize/5);
var pivot:DisplayObject3D = new DisplayObject3D();
pivot.addChild(planet);
pivots.push(pivot); // pivots is a global Array holding all planets
```

Listing 5.3: Adding a texture to planet objects

```
function factorize(p){
 var startTime = new Date();
 var prime = 0;
 var factors = new Array();
 var endCriteria = Math.sqrt(p);
 for (var i=2; i<=endCriteria; i++){
  if (isPrime(i)){ // See previous section about prime generation
   var tmp = p % i;
   if (tmp == 0){
    // i is a prime number
    factors.push(i);
    do{
     p = p / i; // Replace p
     tmp = p % i;
     if (tmp == 0){
      factors.push(i);
     }
    }
    while (tmp == 0);
   }
  }
 }
 var stopTime = new Date();
 var timeElapsed = stopTime.getTime() - startTime.getTime();
}
```

Listing 5.4: Prime-number factorization in JavaScript

```
// Keep iterating until all particles reached the bottom
while (allParticlesReachedBottom() == false){
 // Step through all rows
 foreach (row in rows){
  // Step through all particles of the current row
  foreach (particle in row){
   // If the particle belongs to the very first row...
   if (particle.row == rows[0]){
    if (getChance(3, 7)){
     // ... let the particle start moving at a chance of 3:7
     particle.startMoving();
    }
   }
   // If the current particle does not belong to the very first row...
   else{
    // .. if the forerunner particle has started moving...
    if (particle.foreRunner.hasStarted){
     if (getChance(3, 7)){
      // ... let the current particle start moving at a chance of 3:7
      particle.start();
     }
    }
   }
  }
 }
}
```

Listing 5.5: Algorithm for particle-movement in the 2D-test

```
var randomNumbers = new Array;
var seed = 0; // Initial seed value
var priorIndex = 0;
var numOfNumbers = 1000000; // Generate 1000000 values

// Initialize the generator
var m = 281474976710656;
var b = 29741096258473;
var a = 513;
randomNumbers[0] = seed;

var startTime = new Date();

for (var i=0; i<numOfNumbers; i++){
 var randomNumber = (a * randomNumbers[priorIndex] + b) % m;
 randomNumbers.push(randomNumber);
 priorIndex++;
}

stopTime = new Date();
```

Listing 5.6: Pseudo random key generator (implemented in JavaScript)

5.1.2 Focus-test

```
// The input text as a sequence of bits
var bitstream = getAsBitstream(input);

// Will convert a string to its binary representation
function getAsBitstream(str){
 var res = "";
 for (var i=0; i<str.length; i++){
  /* dec2bin converts a number to its binary representation,
     e.g. dec2bin(5) = 101.
     See the full source on http://www.timo-ernst.net/riabench-start
     for the exact implementation */
  res += dec2bin(str.charCodeAt(i));
 }
 return res;
}
```

Listing 5.7: Converting a string into a bit-stream

```
// The reverted version of bitstream
var bitstream_reverted = revert(bitstream);

/**
 * Reverts a given String. Example: s="Hello". Then revert(s)="olleH".
 * @param str String The string to revert
 * @return String The reverted version of str
 */
function revert(str){
 var ret = "";
 for (var i=((str.length)-1); i>=0; i--){
  ret += str.charAt(i);
 }
 return ret;
}
```

Listing 5.8: Reverting a string in JavaScript

5.1.3 Jaction

5.1.3.1 Various

```
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">
 <head>
  <title>Jaction MD5 demo page</title>
  <script type="text/JavaScript">
  <!--
   function md5(str){
    /* Assuming, there is such a JavaScript function that
       does the actual md5 hashing. See section 2 for the
       full code */
    return toMD5(str);
   }
  //-->
  </script>
 </head>
 <body>
  Welcome to the Jaction md5 demo page.
  <embed src="Jaction_demo.swf" />
 </body>
</html>
```

Listing 5.9: HTML-wrapper for the MD5-example using Jaction

```
start();

function start():void{
 var a = 3;
 var b = 4;
 var result = multiply(a, b);
 doSomething(result);
}

// Calls a webservice synchronously
function multiply(a, b):int{
 var syncWebservice = new SyncWebservice(); // A random webservice

 // Application freezes til method call finished
 result = syncWebservice.multiply(a, b);

 return result;
}
```

Listing 5.10: Example for a synchronous method call

```
start();

function start():void{
 var a = 3;
 var b = 4;

 /* The variable "result" will not be assigned to a value because
    "multiply(a, b)" does not have a return-value. */
 var result = multiply(a, b);

 // This call will fail because "result" is empty.
 doSomething(result);
}

// Multiplies two numbers using an asynchronous webservice
function multiply(a, b):void{
 var asyncWebservice = new AsyncWebservice(); // A random webservice

 // Add event-listener in case the RPC successfully returns
 asyncWebservice.addEventListener(
  AsyncWebservice.successEvent,
  success
 );

 // Add event-listener in case the RPC call fails
 asyncWebservice.addEventListener(AsyncWebservice.failEvent, fail);

 asyncWebservice.multiply(a, b); // Call the service
}

// Gets called when the remote method call successfully returns
function success(result):void{
 /* How to return the value?
    return result; <-- will not work due to interruption of the
    call-stack */
}

// Gets called if RPC call failed (e.g. due to timeout)
function fail():void{
 MessageBox.show("Dear user, the service is currently not available");
}
```

Listing 5.11: Example for an asynchronous method call

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application applicationComplete="init();"
 xmlns:mx="http://www.adobe.com/2006/mxml">

 <mx:Script>
  <![CDATA[
   import mx.controls.Alert;
   import flash.external.*;

   // Will be called on application start
   private function init():void{
    // Call the JavaScript function
    var hash:String = hashWithJaction(strToHash);
    // Show the result in an alert box
    Alert.show("Hashvalue: " + hash);
   }

   // Do a Jaction-call
   private function hashWithJaction(strToHash:String):String{
    return ExternalInterface.call(
     "md5", /* The name of the JS function */
     strToHash /* parameter to pass to the JS function */
    );
   }
  ]]>
 </mx:Script>
</mx:Application>
```

Listing 5.12: Calling external JavaScript-functions from within Flash

```
private function hashWithJaction(strToHash:String):void{
 try{
  // Call the JavaScript function
  hash = ExternalInterface.call(
   "md5", // The name of the JS function
   strToHash // function parameter
  );

  // Check if the Jaction call was successful
  if (!isValid(hash)){ // Assuming there is such a function isValid()
   // Fallback to standard AS3 call (No Jaction)
   hash = MD5.hash(strToHash);
  }
 }
 catch(Exception e){
  // Fallback to standard AS3 call (No Jaction)
  hash = MD5.hash(strToHash);
 }
}
```

Listing 5.13: Strengthened version of the Jaction prototype

5.1.3.2 Final prototype source for Jaction (with MD5 example)

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application applicationComplete="init();"
 xmlns:mx="http://www.adobe.com/2006/mxml">

 <mx:Script>
  <![CDATA[

   /**
    @author Timo Ernst
    http://www.timo-ernst.net
    License: GPL
   */

   import mx.controls.Alert;
   import flash.external.*;
   import com.adobe.crypto.MD5;
   import de.timoernst.jaction.Jaction;

   /**
    * Will be called on application start
    */
   private function init():void{
    var strToHash:String = "Please hash me";
    var hash:String = "";

    if (Jaction.isJactionCompatible()){
     hash = hashWithJaction(strToHash); // Do the Jaction call
    }
    else{
     hash = MD5.hash(strToHash); // Do the normal AS3 call
    }

    // Show the result in an alert box
    Alert.show("Hashvalue: " + hash);
   }

   /* Separates the given string by the separating condition and
      returns the result as an array */
   private function explode(separator:String, string:String):Array {
    var list:Array = new Array();

    if (separator == null) return list;
    if (string == null) return list;

    var currentStringPosition:int = 0;
    while (currentStringPosition<string.length) {
     var nextIndex:int = string.indexOf(separator,
        currentStringPosition);
     if (nextIndex == -1) break;
     var word:String = string.slice(currentStringPosition, nextIndex);
```

```
50        list.push(word);
51        currentStringPosition = nextIndex+1;
52       }
53       if (list.length<1) {
54        list.push(string);
55       } else {
56        list.push(string.slice(currentStringPosition, string.length));
57       }
58
59       return list;
60      }
61
62      /**
63       * Use Jaction for encoding
64       * @param strToHash The input string that is to be hashed
65       * @return The hash representation of strToHash
66       */
67      private function hashWithJaction(strToHash:String):String{
68       var hash:String = "";
69       try{
70        hash = Jaction.call( // Call the JavaScript function
71         "md5", // The name of the JS function
72         strToHash // function parameter
73        );
74
75        if (!isValid(hash)){ // Fallback if the return value is corrupted
76         hash = MD5.hash(strToHash);
77         Alert.show("Return value was invalid.");
78        }
79       }
80       catch(e){ // Fallback if Jaction call failed
81        hash = MD5.hash(strToHash);
82        Alert.show("Jaction call fail: " + e.toString());
83       }
84
85       return hash;
86      }
87
88      /**
89       * Checks if the hash value is empty or null
90       * @param str The hash value to check
91       * @return Returns true if the value was valid. False if not.
92       */
93      private function isValid(str:String):Boolean{
94       if (str == null) return false;
95       if (isEmpty(str)) return false;
96       return true;
97      }
98
99      /**
100      * Checks if the given String is empty
101      * @param str The String to check
```

```
102     * @return Returns true if the String was empty. False if not.
103     */
104    private function isEmpty(str:String):Boolean{
105     if (str.length == 0) return true;
106
107     /* Iterate through the String and check whether it
108        consists of white-spaces only or not */
109     for (var i:int=0; i<str.length; i++){
110      if (str.charAt(i) != " ") return false;
111     }
112     // Return true if the String contained only spaces
113     return true;
114    }
115   ]]>
116  </mx:Script>
117 </mx:Application>
```