IADIS International Journal on WWW/Internet
Vol. 12, No. 2, pp. 131-148
ISSN: 1645-7641

PERFORMANCE EVALUATION FRAMEWORK OF ALL CLASSES OF SELECTORS FOR JAVASCRIPT LIBRARIES

Sotiris P. Christodoulou. *Computer & Informatics Engineering Dept, Technical Educational Institute of Western Greece, Patras, Greece*

Andreas B. Gizas. *HPCLab, Computer Engineering & Informatics Dept., University of Patras, Patras, Greece*

ABSTRACT

Many JavaScript libraries (JL) have become available in order to facilitate programming of rich client-side interactions in web applications. In some cases these interactions may cause non-negligible overhead on the response time of web applications, esp. on average mobile devices, that negatively influence user experience. An important fraction of this overhead is caused by some JL's selectors on DOM elements. The aim of this work is to thoroughly study and evaluate the performance of JL's selectors and categorize them in various performance classes. Towards this purpose, we produced a test-suite of 263 selectors to cover all possible kinds of selectors found in real world cases and evaluate the performance of these selectors on 6 popular JLs, with an enhanced version of slickspeed suite and a large set of jsperf tests. The main result of this work is to introduce a framework (methodology and a set of tools), that can be used by developers to estimate the performance overhead caused by JL DOM selectors in a given web page, discover bad performing selectors and optimize them.

KEYWORDS

Javavascript Libraries selectors, performance evaluation, methodology and tools.

1. INTRODUCTION

Javascript is one of the most popular programming languages today. Along with the growth of demands for more comprehensive and interactive user interfaces, the size and the complexity of web applications are also increasing. JavaScript is also becoming a general purpose computing platform for browsers (Richards et.al., 2010), for office applications (openoffice.org), for RIA frameworks (like Google Web Toolkit, Qooxdoo.org,

Cappuccino.org) and even program development environments (like lively-kernel.org). In recent years Web sites are becoming more accessible by portable and wireless devices. According to Pew Internet[1] in 2012, 55% of Americans use a mobile device to access web. 31% of these mobile internet users say that's the primary way they access the web. Obviously, JavaScript performance is a more critical factor on mobile devices.

Due to the plethora of applications that JavaScript serves and the variety of programming needs, several **JavaScript Libraries (JLs)** have been developed in order to facilitate the work of web programmers. These libraries aim to be a useful tool for simplifying JavaScript code development and reusing blocks of code by writing fewer code lines. Moreover they provide clearer structure, new features, cross-browser support, pre-built applications, ready to use plug-ins. For these reasons, JavaScript libraries, have become most popular[2], and gain an increasingly number of developers nowadays. Many developers have decided to program much of the functionality of their applications using a JL at client side, while using the server primarily to send and receive data. Shifting functionality to the client side enables the potential for a much more powerful and responsive UI of web applications, which has always been an advantage of native apps. But what about the performance overhead added by JLs? Can be considerable?

Let's start by making clear the difference between JS libraries and JS Frameworks. JS Frameworks are bigger, built on top of a JL (most of them on top of JQuery) and provide more advanced tools for developers, like data model as part of an MVC type architecture, and some sort of built-in templating. Popular JS Frameworks include knockout.js, backbone.js, angular.js, ember.js, etc. In this work we focus on the performance of JLs, which influence the performance of JS Frameworks as well. Main aim of a JL is to facilitate programming of rich client-side interactions in web applications. However, in some cases these interactions may cause non-negligible overhead on the response time of web applications, esp. on average mobile devices, that negatively influence user experience. The main scope of this work is to thoroughly study and evaluate the performance of JL's selectors, categorize them in various performance classes and accurately estimate their overhead on various devices and different browsers.

This research work may seem like a lot of effort to spend on a seemingly tiny performance differential. However, research has consistently shown a strong correlation between fast sites and higher conversion rates, more user actions per visit, and user satisfaction.

- 37% of consumers will shop elsewhere if a mobile site or app fails to load in 3 seconds (Harris Interactive, 2013).
- The abandonment rate for mobile shopping carts is 97%, compared to 70% for desktop carts. Performance is a significant abandonment factor (Google I/O Keynote, 2013).
- A company's business performance suffers when its Web page takes longer than three seconds to load, according to a study by Aberdeen Group (Aberdeen Group, 2012). An additional delay of even one second can result in a loss of 7 percent of customers, an 11 percent decline in page views and a 16 percent drop in customer satisfaction.

The paper is organized as follows. After discussing related work, we select the JLs under evaluation. In section 4 we introduce a methodology and a set of tools for dynamic analysis of JL-based web applications and produce statistics on the volume and the type of selectors actually executed. Furthermore, we extensively analyze the selectors used in a set of real

[1] http://www.pewinternet.org/2012/06/26/cell-internet-use-2012/

[2] The State of Web Development 2010 - Web Directions Survey, http://www.webdirections.org/sotw10/

applications, classify them according to their syntax and their performance and produce a test-suite of more than 260 selectors to cover most possible kind of selectors found in real world cases. Based on this test-suite we evaluate the performance overhead of selectors on six popular JLs, by constructing and executing an enhanced version of slickspeed suite (described in section 5) and a large set of jsperf tests (section 6). The results of these tests are also presented and discussed in these sections, where we classify all kind of selectors in six discrete classes according to their performance. In section 7 we present a methodology that developers can use to evaluate the performance of their selectors and discover which should be optimized. This work is completed with a summary of the conclusions and the intended future work.

2. RELATED WORK AND MOTIVATION

The heterogeneity of client devices / web browsers and the complexity of web applications' front-end, lead to increased development and test efforts. The performance evaluation of such applications is an important success factor, but measuring front-end performance requires a deep understanding of measurement tools and techniques as well as a lot of human effort. Recently, researchers (Westermann et.al., 2013) proposed an approach with which developers can assess front-end performance without actually measuring it, but by using prediction models. Towards this direction, our work may help such models to predict more accurately the performance overhead of web apps on mobile devices, based on which JL is used, the DOM size and the selectors executed during run-time.

There are few research efforts on comparing or evaluating Javascipt Libraries or Frameworks and they mainly focus on comparing JLs' features, like multimedia support on developing RIA (Rosales-Morales et.al., 2011). On the other hand there are a few articles on reputable web sites of technology companies or well-known Javascript developers that present some sort of comparison of JLs or JFs, like:

- http://www.ibm.com/developerworks/library/wa-jsframeworks/wa-jsframeworks-pdf.pdf
  Lennon Joe, Compare JavaScript frameworks: An overview of the frameworks that greatly enhance JavaScript development. IBM 2010.
- http://www.codefessions.com/2012/04/mobile-javascript-frameworks-evaluation.html
  Mobile JavaScript frameworks, Evaluation and Comparison (Apr 2012)
- http://www.softfinity.com/blog/category/javascript-libraries/
  The Battle of Modern Javascript Frameworks (Apr 2013)

These articles provide some good practices for developers, but are mostly compact, evaluate JLs generically and lack of performance evaluation. Few online articles (like Grabner Andreas, 101 on jQuery Selector Performance (Nov 2009) http://apmblog.compuware.com /2009/11/09/101-on-jquery-selector-performance/ and jQuery Performance Rules (Apr 2009) http://www.artzstudio.com/2009/04/jquery-performance-rules/) are coping with the performance evaluation of the core components of JLs, namely DOM traversal (selector engine) and DOM manipulation. However, by reading out the comments on these articles and studying a plethora of jsperf.com tests, we understand that an in-depth comparison of these libraries, esp. regarding their performance overhead on average mobile devices, would interest the mobile web developers working with JLs. Moreover, an article written by Steve Souders on Performance Impact of CSS Selectors (Performance Impact of CSS Selectors, Mar 2009

http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/) motivates us to start our study by primarily focusing on the performance of the selector engines of JLs and leaving DOM manipulation performance for future work.

3. JAVASCRIPT LIBRARIES UNDER EVALUATION

A JavaScript Library typically provides a library of classes or functions that implements a multitude of operations like managing DOM traversal and manipulations, insert visual effects, managing Ajax manipulations and layout, whilst at the same time impose an architecture that provides ways to extend the library (e.g. plug-ins, modules etc.).

As stated, our study is focused on the performance of the JL core component, namely DOM Traversal / Selector Engine, while planning to work on DOM Manipulation performance in future work. Let's define the functionalities of these two components under study:

1. **DOM Traversal and Selectors Engines:** All JLs implement a mechanism for easier element(s) selection. These selectors make the process of obtaining references to HTML elements much easier, and allow developer to select element(s) by ID, class name, element type, hierarchy or by using a series of pseudo-selectors.
2. **DOM Manipulation:** Various manipulations on DOM elements like add, hide, remove and copy elements, change their properties such as color, width, height, etc.

JavaScript developers can choose among a variety of JS Libraries. 62%[3] of web sites are using at least one JL. Today, the most popular JavaScript Library is jQuery, but there are some other similar JLs that developers could also use like: MooTools, Prototype, YUI, ExtJS and Dojo. Besides the latest version of each JL, we have also evaluated some older versions that are still running on a significant number of web sites (as developers either don't maintain applications any more or they are afraid upgrading due to compatibility issues). Fig.1 shows which versions of JLs we have evaluated and their usage percentage.

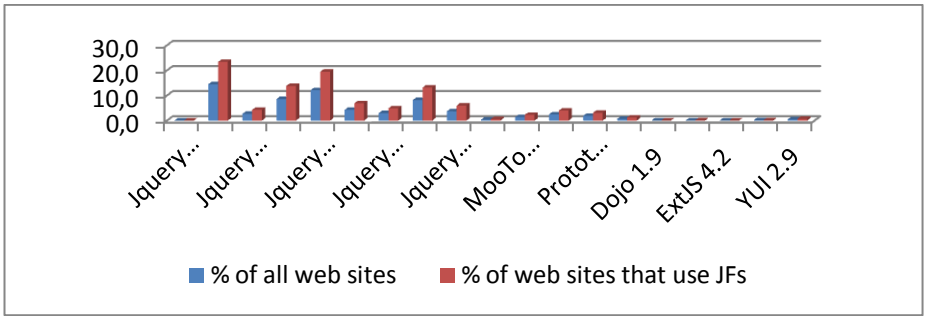

Figure 1. Javascript Libraries usage percentage [3] (Aug2014)

[3] http://w3techs.com/technologies/overview/javascript_library/all

4. JL DOM SELECTORS

All JLs provide a set of DOM selectors. Most of these selectors were borrowed from CSS 1–3 specifications, while each JL provide some additional selectors (some of them are common among JLs) offering a powerful set of tools for matching a set of elements in an HTML page. Selectors may range from simple element names to rich contextual representations. We classify selectors according to their type, as described in W3C Specification of Selectors in CSS3[4]. Table 1 presents this classification.

Table 1. Selector Types

Simple Selectors				
Universal Selector	*			
Tag Selectors	tag			
ID Selectors	#id	tag#id		
Class Selectors	.class	tag.class		
Attribute Selectors	[att]	[att=val]	[att|=val]	etc.
	tag[att]	tag[att=val]	tag[att|=val]	etc.
Multiple Attribute Selectors	[att1][att2]	[att1=val1][att2|=val2]		etc.
	tag[att1][att2]	tag[att1=val1] [att2|=val2]		etc.
UI element States Pseudo-classes	:enabled	:disabled	:checked	
Structural Pseudo-classes	:root	:empty		
	:nth-child(val)	:nth-of-type(val) etc.		
The negation pseudo-class	:not(selector)			
Other pseudo-classes	:target	:lang(val)		
Combinators				
Descendant combinator	selector1 selector2			
Child combinator	selector1 > selector2			
Adjacent sibling combinator	selector1 + selector2			
General sibling combinator	selector1 ~ selector2			
Group of selectors				
Multiple Selectors	Selector1, selector2,selector3 …			

4.1 Dynamic Analysis of JL Selectors executed in Real Web Applications

An important issue in our study was to discover what kind of selectors developers actually use in their programs, and highlight patterns and programming practices used, either good or bad. To achieve this we used Dynatrace AJAX Edition tool of Compuware Inc. It is a freeware live web performance diagnostics tool for JavaScript execution, DOM access, rendering activities and network traffic analysis. The tool provides data on Performance, User Experience, Path

[4] Selectors Level 3, W3C Recomm. 29/9/2011 http://www.w3.org/TR/css3-selectors/

analysis of JavaScript execution during runtime, Network and HotSpots analysis. HotSpot analysis includes, among others, which selectors were actually executed either within the page loading phase, or when the user interacts with the page. This is important for our case, as different set of selectors can be executed for each distinct use case of applications under evaluation. Dynatrace also measures the execution time for each selector, but it can be only used for desktop computers and Firefox or Internet Explorer. Thus, we use it within our methodology, just to extract the selectors of an application.

We analyzed 48 jQuery interactive applications found in GitHub. For each application we traced one minute interaction with it by using Firefox. Afterwards, in HotSpots Analysis, we filter all *Contributors* that contain the *init(")* function within their text, as init() is the jQuery function that every selector calls whenever a new instance of jQuery() starts in order to interact with the DOM. This process produced a list of *1979 jQuery selectors* that developers actually use in their programs and produced after real user interaction. Fig. 2 shows the top 20 patterns of jQuery Selectors used.

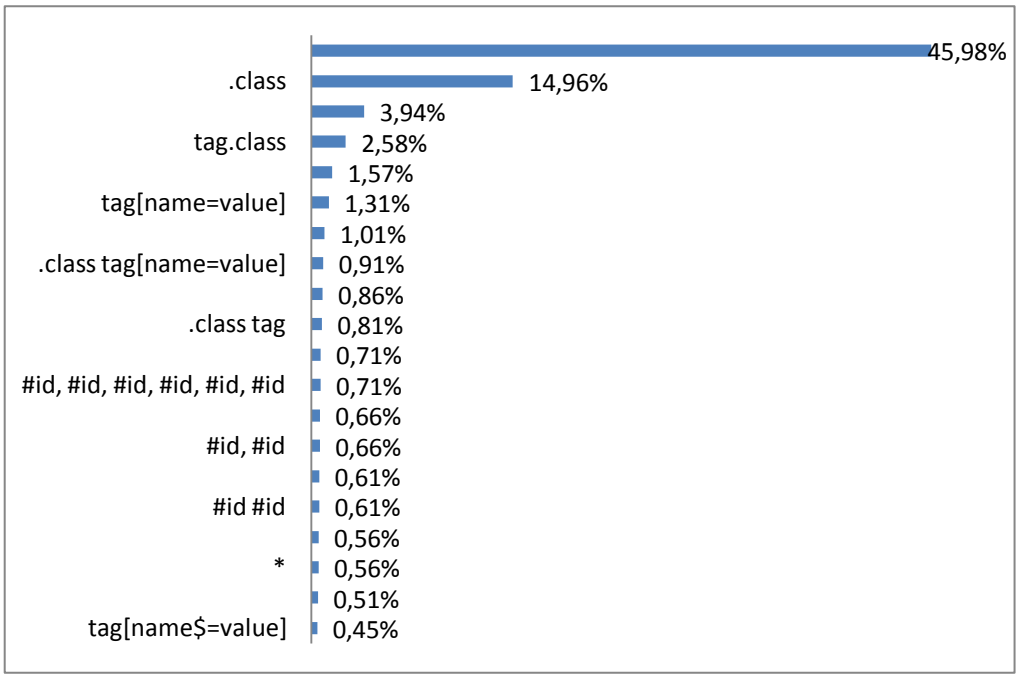

Figure. 2. Patterns of jQuery Selectors used

4.2 JL Selectors Test-Suite

Based on the results of the abovementioned analysis and the results of a similar analysis (Selectors that People Actually Use, Feb 2008, http://ejohn.org/blog/selectors-that-people-actually-use/) provided by John Resig (JQuery creator), we constructed a test-suite of 263 selectors based on 131 distinct selectors' patterns, that cover more than 98% of all possible kind of selectors found in the real world cases analyzed. Moreover, we have included some

selectors not found in real applications, but can assist us in statistical analysis of the results. For instance we have included the selector *[type=checkbox]*, even if we didn't discover it, in order to be able to compare the performance of the equivalent selector *:checkbox*.

We had to group these selectors together, in such a classification that will be helpful on analyzing the results. The grouping of simple selectors is straightforward, but we need some rules to classify complex selectors containing combinators, esp. the descendant combinator which is by far the most commonly used. Our decision was based on the fact that most JLs selector engines are parsing the selectors from right to left. Thus the most right sub-selector determines the group of the complex selector. E.g. selector *tag tag.class* is classified under the Class Selectors group. Table 2 presents the Selector Groups of our Test-suite. These selectors were carefully produced for a local copy of this page: W3C specification of Selectors in CSS3[4]. We have modified this copy in various ways, in order to be able to cover all selectors we want, e.g. we add a form at the end of the page, for testing form selectors, like *:checked*. We tried to include selectors that match various numbers of elements, and even none elements, in order to understand the overhead of such "bad" selectors. Some selectors match elements that are dispersed over the DOM, while others match elements clustered in a certain part of the DOM tree. For ID selectors we have included selectors that query for ids in different part of DOM tree (top, middle, and bottom) and in different depth.

The full-list of the selectors in our Test-suite can be accessed by downloading the SlickSpeed-enhanced tool (see next section) that we built based on the Slickspeed[5] tool developed by Mootools team.

Table 2. Selector Groups in our Test-suite

	214 Common Selectors supported by all JLs
ID Selectors	#id tag#id #id #id[6] tag#id tag#id
Class Selectors	.class, tag.class *and all complex selectors where the right-most sub-selector is a class selector.*
Tag Selectors	*, Tag *and all complex selectors where the right-most sub-selector is a tag selector*.
Attribute Selectors	[att],[att=val],[att|=val],tag[att],tag[att=val],tag[att|=val],[att1][att2], [att1=val1][att2|=val2] tag[att1][att2],tag[att1=val1][att2|=val2], etc. *and all complex selectors where the right-most sub-selector is an attribute or multiple attribute selector.*
Child Selectors	:nth-child(val) :nth-of-type(val) etc.
Other Selectors	:enabled, :disabled, :checked, :empty, :not(selector), :contains(text)
	49 JQuery Extensions Selectors (some are supported by other JLs)
JQuery Extensions Selectors	[att!=val], :selected, :checkbox, :input, :visible, :hidden, :header, :first, :last, :eq(val), :gt(val), :lt(val), :even, :odd, :parent, :has(selector), *some context selectors like $(selector, context), and all complex selectors that include at least one of the above selectors*

[5] https://github.com/kamicane/slickspeed

[6] Although is meaningless to use the descendant combinator with an #id at the end, we included such selectors, as we found out that they are used in some programs.

5. SLICKSPEED-ENHANCED

Slickspeed is a simple test-suite for speed / validity tests for DOM selectors in JLs. Based on Slickspeed we developed a new improved version called Slickspeed_Enhanced[7] that additionally provides:

- Saving the results to the server through a form, an important feature when testing from mobile devices.
- A much bigger and organized set of 263 predefined selectors, with diverse complexity, derived from the analysis in previous sections.
- Users can run their tests against three different HTML files with varying DOM size. Trends (http://httparchive.org/trends.php) show that a medium DOM size page consists of about 1000 DOM elements. Thus we decided to use three indicative sizes, i.e. small (140 elements), medium (902 elements) and large (2068 elements). We carefully produced the three test HTML pages, so that the majority of selectors are matching elements even in the small one, obviously in fewer numbers. You can see the matched elements per DOM and per selector in the results of slickspeed enhanced tests in our site.
- Finally, a major addition we provide is a new **Slickspeed_Commands** implementation, where testers can measure the time overhead not only for selectors, but for one JL command or a piece of code. We used this tool to compare the performance of 35 (out of 49) jQuery Extensions Selectors with equivalent jQuery commands by using jQuery functions.

5.1 Performance Tests with Slickspeed-enhanced

The tool measures the execution time for each selector and the total time for all selectors. We used the core compact versions of all libraries, except Prototype that does not provide one. All files with the detailed measurements and information about the tests described below are available under this web page: http://alife.hpclab.ceid.upatras.gr:100/JLmetrics/. Our test bed includes the configurations below:

- **PCs/Laptops:** We have tested the performance of each JL under 3 major browsers (Firefox, IE11, and Chrome), on 2 strong computers. The overall outcome is that performance is not an important concern when we run JL apps on desktop or laptop computers.
- **Mobile Devices:** In mobile devices rendering and execution times of JavaScript code are much higher because of hardware capabilities of those devices and mobile versions of browser engines. We conducted the same performance tests with some indicative mobile sets (devices, operational systems, browsers), shown in Table 3.

Before every test, cache memory was cleared. During the tests, no other applications were running and there was no user interaction (like scrolling, zooming, etc.). The devices were operated only with their batteries.

[7] https://github.com/gizas/Slickspeed_Enhanced

Table 3. Mobile devices configurations

Brand / Name	OS	CPU	Browsers
Samsung Galaxy S3 mini	Android OS, v4.1.2	Dual-core 1 GHz Cortex-A9 with 1GB RAM	Android Browser 2.3.3
Nokia Lumia 820	Win Mobile 8.0	Dual-core 1.5 GHz Krait with 1GB RAM	Internet Mobile 10
Apple iPhone 5	iOS 7.0.4	Dual-core 1.3 GHz Swift (ARM v7) with 1GB RAM	Safari Mobile 7.0

The first observation by analyzing the results is that some selectors match different number of elements when used with different JL. For instance, :contains pseudo-class selector works differently with Dojo and Prototype 1.6.1 (with Prototype 1.7.1 works correctly), while child selectors are not working with early versions of MooTools and Dojo matches more elements than it should. Fortunately, the latest versions of JLs are working correctly for almost all 214 common selectors. Thus, we present and compare the results of only the latest versions of JLs, apart from jQuery that we show the results for 8 different versions of the most popular library. Figure 3 illustrates the total execution times for each JL, on each mobile browser, on each device. Table 4 presents the slowest common selectors on Big DOMs per JL.

Figure 4 shows the total execution time for all jQuery versions, for the JQuery Extensions Selectors and equivalent jQuery code (see Slickspeed_Commands tool described above). jQuery 1.3.2 doesn't support some of these selectors, thus we exclude it from the comparison. Table 5 presents the slowest jQuery Extensions Selectors on Big DOMs per jQuery versions.

We measured much higher execution times than in PCs (from 10x to 40x depending on the device). The execution times are mainly depending on the CPU power, thus we cannot compare the performance of different browsers running on different devices, but we can compare the performance of different JLs on the same browser. The main conclusions on these performance results are:

- Browsers on slower devices display much higher execution times for all JLs.
- ExtJS is the fastest in all test sets. This is more obvious on average devices.
- Dojo performs poorly in all test sets. The difference with other JLs is bigger on average devices.
- MooTools and YUI perform much slower on IE 10 Mobile, while on the other browsers their execution times are close (or even better) to the ones of other JLs.
- While Android browser performs worse than IE10 Mobile on general selectors, it performs better for selectors based on JQuery Extensions.
- For selectors based on JQuery Extensions, we observe that newer versions are slower than early versions.
- The execution times are analogous to DOM size.

Additionally, we repeated all performance tests, with heavy user interaction during the tests, i.e. scrolling, zooming in and out, etc. We measured much worst times (from 4x to 20x times), except iPhone, that shows not affected by user interaction. For other devices, assuming that users are not totally idle when they are browsing, we can estimate that the correct performance times are the double of the values that are presented in this paper. Thus, the main conclusion is that mobile users on average mobile devices are probably having bad experience with some JL applications that include bad selectors that executed repeatedly.

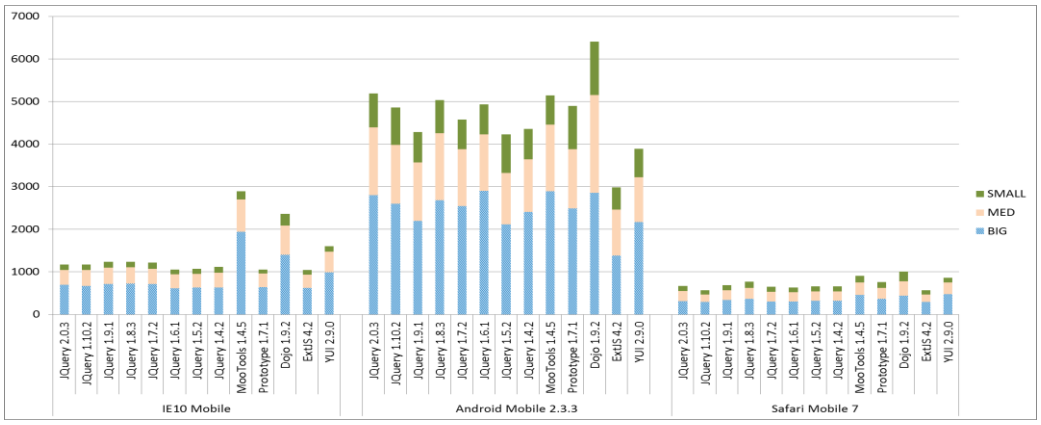

Figure 3. Performance of JLs Common Selectors on Mobile Browsers per DOM size (ms)

5.2 General Conclusions about jQuery Extensions Selectors

By further analyzing the results per selector, some specific selectors came out to be really slow in performance. A surprising result is that some selectors are much slower in newer versions of jQuery. Another important outcome, shown in Table 5, is that :has selector is by far faster than .has() function, while jQuery documentation recommends the use of .has() function. Such big latencies may cause halting of mobile browsers. Other selectors with big latency include :hidden and :visible jQuery selectors, which are commonly used by developers. Moreover, the use of jQuery context selector is slower in comparison to the equivalent descendant selector. Other selectors that users should avoid is :not(), :contains(selector), or [colspan=val] attribute selector.

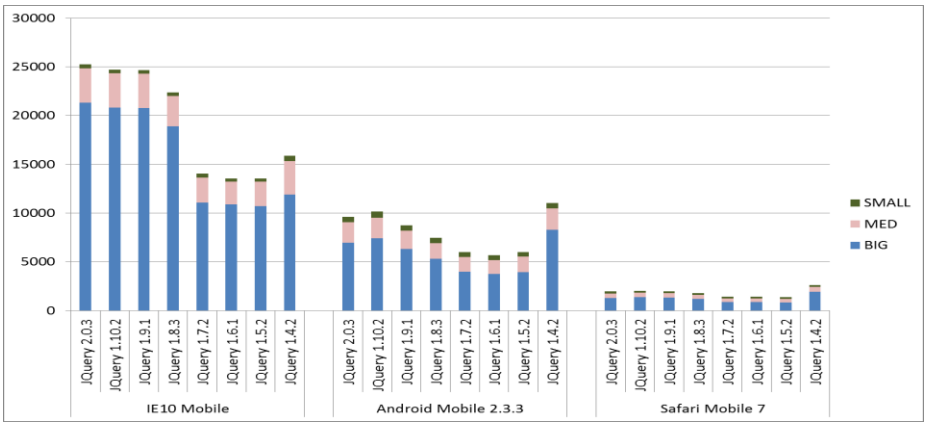

Figure 4. Performance of Jquery Extensions Selectors on Mobile Browsers per DOM size (ms)

Table 4. Slow common selectors on Big DOMs per JL (msec)

Selectors	JQ 2.0.3	JQ 1.10.2	JQ 1.9.1	JQ 1.8.3	JQ 1.7.2	JQ 1.6.1	JQ 1.5.2	JQ 1.4.2	MT 1.4.5	Proto 1.7.1	Dojo 1.9.2	Extjs 4.2.0	Yui 2.9.0	Browser
	70	73	122	111	119	49	50	54	56	59	32	39	84	IE 10
[colspan=2]	118	176	192	177	320	346	184	259	62	155	56	46	104	Andr
	18.5	17	27	26	22	13.5	13.5	15	14.5	14	11.5	14	21	Safari 7
	22	23	21	29	8	10	11	12	30	11	3	4	153	IE 10
ul:not(ul ul) > li	81	116	52	141	96	73	65	80	61	71	39	30	236	Andr
	9.5	8.5	9	8.5	8.5	9	8	9	5	8.5	3	6.5	34	Safari 7
div[class!= made_up]	7	7	8	9	6	3	3	3	8	3	3	4	6	IE 10
	84	39	21	33	29	21	12	26	15	44	4	34	10	Andr
	3	3	5	5	2.5	2	1.5	1.5	2	1.5	1	5	4	Safari 7
	1	1	1	1	1	1	2	2	1	2	17	1	2	IE 10
:checked	122	82	65	76	5	4	4	5	110	4	118	2	116	Andr
	0.5	0.5	1.5	1.5	1	1	2	1.5	1	1.5	9	0.5	21	Safari 7
:contains(elector)	137	115	118	125	129	117	132	125	224	129	240	115	283	IE 10
p:contains(elector)	15	19	16	18	16	16	17	16	42	18	29	13	39	
:contains(elector)	187	201	142	187	181	234	178	154	291	101	331	88	445	Andr
p:contains(elector)	57	20	40	92	52	51	53	54	48	94	50	30	63	
:contains(elector)	25.5	25.5	24.5	23.5	24.5	25	25.5	29	107	25.5	49.5	24.5	103	Safari 7
p:contains(elector)	5.5	3.5	3.5	3.5	4	3.5	6	6	13.5	6	7	5.5	21	

Overall, Slickspeed enhanced is a tool that facilitates the concurrent comparative evaluation of several versions of JLs at once. But for precisely estimating the performance of each class of selectors, we needed a more statistically significant tool. We chose jsperf for this purpose.

6. PERFORMANCE TESTS WITH JSPERF

JSPerf is a service aim to provide an easy way to create and share test cases, comparing the performance of different JavaScript snippets by running benchmarks. jsPerf is based on Benchmark.js, a robust JavaScript benchmarking library that works on nearly all JavaScript platforms, supports high-resolution timers, and returns statistically significant results. However, we still need Slickspeed Enhanced for two reasons: a) JSPerf results are not accurate when the execution time is close to 1 sec (as returns operations/sec) and b) comparing several JLs is very hard, as we have to reproduce all tests for each different version of each JL and rerun the tests many more times.

Table 5. Slow jQuery Extensions Selectors on Big DOMs per jQuery versions (msec)

Selectors	JQ 2.0.3	JQ 1.10.2	JQ 1.9.1	JQ 1.8.3	JQ 1.7.2	JQ 1.6.1	JQ 1.5.2	JQ 1.4.2	Browser
$(":visible")	319	325	325	345	505	509	502	1090	IE10 mob
	36	71	37	53	49	47	49	136	Andr 2.3.3
	11	13	15	14	15	15	15	35	Safari 7 mob
$(":hidden");	293	294	304	294	500	506	503	1044	IE10 mob
	35	71	38	23	40	38	39	106	Andr 2.3.3
	9	11	9	9	11	11	12	26	Safari 7 mob
$(":has(code)");	458	456	442	485	449	488	449	535	IE10 mob
$("*").has("code");	16160	15660	15560	13817	6064	6015	5963	4806	
$("p:has(code)");	99	86	92	96	88	93	91	110	
$("p").has("code");	1869	1863	1891	1624	859	841	841	874	
$(":has(code)");	188	245	241	339	343	272	270	531	Andr 2.3.3
$("*").has("code");	4293	4320	4103	3192	1803	1822	1772	5176	
$("p:has(code)");	112	89	52	54	53	46	45	84	
$("p").has("code");	1000	841	492	452	256	246	423	709	
$(":has(code)");	38	40	40	39	39	40	38	130	Safari 7 mob
$("*").has("code");	808	874	838	701	371	395	397	1169	
$("p:has(code)");	9	10	11	10	10	10	10	26	
$("p").has("code");	68	73	69	55	25	27	25	148	
div p	2	2	2	2	3	2	2	2	IE10 mob
$("p","div");	49	46	59	46	54	53	51	51	
div p	2	3	2	9	2	2	3	6	Andr 2.3.3
$("p","div");	139	181	84	30	34	34	37	70	
div p	1	1	1	1	1	2	1	1	Safari 7 mob
$("p","div");	12	13	13	6	6	7	5	9	

6.1 Tests in JSPerf

In total we have created 360 test cases (presented in Table 6), grouped under 5 tests, classified according to the selector groups of our Test-suite (Table 2) and some extra complex selectors we produced, in order to confirm some conclusions, after analyzing the abovementioned results. We have built and run the above tests for the big, medium and small DOM, on all devices and browsers used in slickspeed tests. To access these tests and their results follow the links shown in the first column of Table 6.

Table 6. Selector Groups in our Test-suite

jsperf.com tests	DOM Size	Description
214 Common Selectors supported by all JLs		
Jsperf.com/a-f-tests	Big	ID, .class, tag.class, tag selectors (with descendant combinator), tag selectors (with other combinators), Multiple selectors
Jsperf.com/a-f-tests/2	Medium	
Jsperf.com/a-f-tests/3	Small	
Jsperf.com/g-j-tests	Big	Attribute selectors, Child Selectors, Other selectors, *Complex selectors found in real web sites and belonging in various groups*
Jsperf.com/g-j-tests/2	Medium	
Jsperf.com/g-j-tests/3	Small	
49 JQuery Extensions Selectors (some are supported by other JLs) equivalent code (by using jQuery functions) for 35 of the above selectors		
Jsperf.com/klmnop-tests	Big	[att!=val], :selected, :checkbox, :input, :visible, :hidden, :header, :first, :last, :eq(val), :gt(val), :lt(val), :even, :odd, :parent, :has(selector), context selectors like $(selector, context), *Complex selectors found in real web sites and include at least one of the above selectors*
Jsperf.com/klmnop-tests/2	Medium	
Jsperf.com/klmnop-tests/3	Small	
20 basic Javascript functions		
Jsperf.com/q-tests	Big	document.getElementById()
Jsperf.com/q-tests/2	Medium	document.getElementsByTagName()
Jsperf.com/q-tests/3	Small	document.getElementsByClassName()
42 complex selectors		
Jsperf.com/r-tests	Big	We produced 42 more selectors than Slickspeed Enhanced. We needed them to confirm some conclusions, after analyzing the first results.
Jsperf.com/r-tests/2	Medium	
Jsperf.com/r-tests/3	Small	

The performance times of the selectors, are different on various browsers and devices, but they are proportional showing that slow selectors perform the same despite the browser or mobile device. According to their performance we categorized the selectors in six categories named A-Selectors (faster selectors) to F-Selectors (slower selectors). Table 7a and 7b shows part of jperf tests results, for all DOMs for Android Browser 2.3.3. The results for the other browsers are proportional. In first column we categorize each selector according to each performance. Below we describe each class of selectors and discuss them.

A-Selectors:

For selecting only one element, the faster way is by using an #id selector. The reason is that it maps directly to a native JavaScript method, getElementById(). Although, #id is 12x slower than the native method, it is still very fast, thus it is safe to be used. Most selectors used in today jQuery programs are of this type. Especially in IE10, the #id is very faster than tag#id, #id #id or #id followed by a pseudo-class, e.g. #id:hidden.

Table 7a. Part of jsperf tests results for Android Browser 2.3.3 (msec)

Class	*SELECTOR*	*Test(s)*	*Small*	*Med*	*Big*
	getElementById	Q01-Q05	0.0005	0.0005	0.0005
A	#id	A01-A05	0.01	0.01	0.01
A	tag#id	A07-A09	0.14	0.13	0.12
A	#id:hidden	L13	0.31	0.33	0.19
	getElementsByClassName	Q14-Q16	0.0010	0.0009	0.0009
B	.class	B01-B03	1.08	6.03	35.71
B	tag.class	C01-C03	2.64	8.29	15.28
	getElementsByTagName	Q07-Q12	0.0246	0.0244	0.0242
B	tag	D01-D06	1.62	5.41	12.74
B	tag,tag	F01-F04	2.94	10.09	16.97
B	#id,#id	F05	2.50	7.41	14.93
B	.class,.class	F06-F07	3.60	11.49	20.64
B	#id,tag,.class	F11	4.95	11.49	21.74
B	[name]	G01-G04, J14	4.20	13.99	29.00
B	[name=value]	G06-G08	3.42	10.57	21.94
B	tag[name]	G11-G14	3.25	11.00	21.43
B	tag[name=value]	G16-G18	3.16	8.59	16.90
C	*	J01	6.94	19.23	41.67
C	:empty	I19	3.80	13.33	29.41
B	SEL:empty	I20	2.22	6.90	14.29
C	:not(SEL)	I11	5.95	23.26	45.45
B	SEL:not(SEL)	I12-I13	2.47	6.74	13.71
C	SEL *	J02-J04	4.98	16.13	31.59
D	$('p','div')	O01	8.70	34.48	125.00
D	$('code','p')	O02	2.75	18.52	90.91
B	$('a','dl')	O03	0.16	7.35	13.70
B	div p	D19	2.48	7.30	14.49
B	p code	D20	2.71	7.30	14.29
B	dl a	D21	2.63	7.30	14.49
F	:checked	I01	24.39	66.67	142.86
B	input:checked	I02	2.16	6.33	13.51
F	:visible	L01	21.28	62.50	125.00
B	input:visible	L05	2.07	6.85	12.35
F	:hidden	L09	18.87	62.50	125.00
B	input:hidden	L14	2.02	6.67	13.51
F	header	M01	15.38	52.63	111.11
F	[colspan=2]	G09	40.00	166.67	333.33
F	[maxlength=20]	R06	38.46	200.00	333.33
B	th[colspan=2]	G18	3.95	8.26	15.15
F	[class=left][type!=radio]	R10	58.82	250.00	500.00
F	[type!=radio][class=left]	R11	40.00	166.67	333.33
F	:contains(elector)	I14	58.82	250.00	500.00
B	p:contains(elector)	I15	2.60	8.93	21.28
F	:has(code)	N03	166.67	500.00	>1,000
B	p:has(code)	N05	3.00	13.89	33.33
F	$('p').has('code')	N06	3.39	66.67	1,000.00
B	div:has(label)	R22	6.99	11.63	22.73
A	#myBody div:has(label)	R23	2.01	3.39	6.41
A	#myDiv div:has(label)	R24	1.64	1.87	2.06

Table 7b. Part of jsperf tests results for Android Browser 2.3.3 (msec)

Class	*SELECTOR*	*Test(s)*	*Small*	*Med*	*Big*
E	.myClass .myForm div:has(input[name!=T3])	P07	66.67	76.92	111.11
	$('.myClass .myForm div').has('input[name!=T3]')	P08	13.33	19.23	25.00
E	$('div[id!=myBody]:contains(not):has(label)');	R37	8.55	20.41	52.63
	$('#myBody div[id!=myBody]:contains(not):has(label)');	R38	2.36	5.68	10.87
	$('#myDiv div[id!=myBody]:contains(not):has(label)');	R39	1.83	2.15	2.43
E	$('div[id!=myBody]:contains(not):has(label) *');	R40	200.00	1,000.00	>1,000
	$('#myBody div[id!=myBody]:contains(not):has(label) *');	R41	33.33	200.00	500.00
	$('#myDiv div[id!=myBody]:contains(not):has(label) *');	R42	26.32	125.00	333.33

B-Selectors:
A selector is slower when we need to get more than one element. The simplest way to do this is by a .class selector or a tag selector, which they also map directly to native JavaScript methods. A .class selector is slower when the DOM size is big, thus in big DOMs prefer to use tag.class instead.

Attribute selectors are 1.5x slower that tag and .class. Use a tag in front of them. Multiple attributes and other operators than equals (=) are not affecting the time. However, there are some exceptions that we analyze below in F selectors.

Prefer Multiple Selectors when possible, esp. on big DOMs, because the DOM is traversed once for all selectors, thus the total execution time is not the sum of the separate times, but the execution time of the slowest selector. Do not use Multiple Selectors for #id, e.g. #id, #id.

All combinatory selectors, no matter how complex they are, as soon as they don't include any F-selector, are belonging to this class of performance. The majority of selectors belong to this category and for the rest of the text we use the keyword SEL for referring to a B-Selector.

C-Selectors:
The universal selector (*) is 4x slower than a B-Selector. Avoid this selector, esp. on big DOMs. SEL * type of selectors are also 2x slower than a B-Selector. Two more selectors (:not() and :empty) belong to this class too. Prefer to use them with a B-Selector in front.

D-Selectors:
jQuery supports context selectors, i.e. the selection of elements within a context. For instance, programmer can use this expression $('p','div') to selects all p elements included within a div. This is equivalent with the selector $('div p'). Context selectors are much slower depending on the matched elements, while B-selectors like $('div p') are faster independently of the matched elements, thus they should be preferred.

F-Selectors:
These are the slowest selectors and developers should always avoid the use of them. Most of them are jQuery Extensions. Avoid using these selectors even in small DOMs.

- :checked, :checkbox, :button, :file, :focus, :image, :input, :password, :radio, :reset, :selected, :submit, :text, :visible, :hidden, :header, :parent, :first, :last, :eq(), :gt(), :lt(), :slice(), :odd, :even, :animated, all child filter pseudo-classes, etc.
- [name=value], when name is one of these keywords: tabIndex, readOnly, maxLength, cellSpacing, cellPadding, rowSpan,colSpan, useMap, frameBorder, or contentEditable.
- [name!=value]
- :contains()
- :has()

E-Selectors:
E-selectors are complex selectors combine C, D and F selectors. They can be very slow when the selector or a part of it, matches high number of elements or it is incorporating more than one F Selectors in combination. Developers should study carefully the performance of these selectors and modify them if it is necessary. Normally, the easiest way to optimize such selectors is to minimize the DOM area that you apply such selectors, by descending from the closest parent ID (if there is no such ID, create one). Some examples can be seen in Table 7b. Especially since a web page is likely to grow more complex over time, the data clearly shows that selectors which don't directly descend from an ID should always descend from an HTML tag when possible.

7. METHODOLOGY

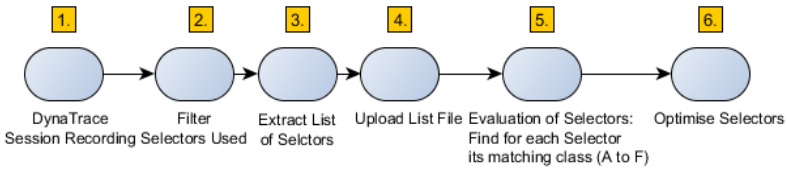

Figure 5. Performance of jQuery (msec)

Based on the results of our test-bed, we introduce a step-by-step methodology that can be used by developers to evaluate the performance overhead caused by the DOM selectors in their web pages, by the use of JavaScript Libraries. This methodology consists of 5 steps (see fig 5), demonstrated below:

- **1st step- Dynatrace Session recording:** User records for a specific time period his or hers activities in a web page with the Dynatrace Ajax Edition tool. Then user navigates and interacts in the page and stops recording at some point.
- **2nd step- Filtering of Selectors used:** Open the session saved from previous step. Locate Hotspot tab in the open menu. Add a new filter like the example image below in order to extract only the init() functions triggered during the session.

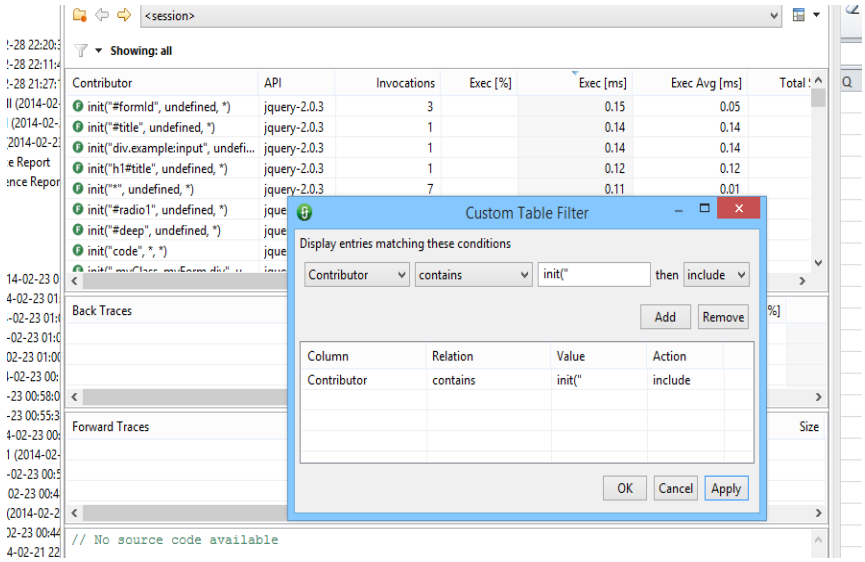

Figure 6. DynaTrace Screenshot

Then a new list of functions will appear, with time execution statistics and number of invocations of each function.

- **3rd step- Extraction of Selectors' List:** Select all the init(")function list and copy-paste them in an excel file. Keep only "Contributor" and "Invocations" columns and delete other unwanted columns. Save this file with a csv extension.
- **4th step- Upload List to our Server:** Upload csv file to server with the help of the form provided at the http://alife.hpclab.ceid.upatras.gr:100/JLmetrics/.
- **5th step- Evaluation of Selectors:** The system handles the uploaded list file finds for each selector its matching class (A to F), and presents the results to the developer.
- **6th step- Optimize Selectors:** Replace D-Selectors with the equivalent B-selector. Replace the F-Selectors with a B-selector. For instance, instead of :checked, give an ID to your form and use the B-selector #formID :checked. Carefully check the performance of every E-selector, esp. those that match many elements, and optimize them mainly by narrowing the DOM area that the selector applies.

8. CONCLUSIONS & FUTURE WORK

The importance of good performance and usability testing is arguably more important now than ever before, especially given the growth of mobile devices. When web applications are as likely to be viewed on tablets or smartphones as on traditional PC browsers, the need for good testing to ensure they will run efficiently is very important. Research proves that a performance difference of a few dozen milliseconds is a user-perceptible delay.

In this work we introduce a step-by-step methodology and a set of tools that can facilitate developers to analyze and evaluate DOM selectors' performance of JavaScript Libraries. This methodology aims to help in the categorization of selectors used into different performance classes and reveal the selectors that developer should optimize by priority. Moreover, by conducting some specific tests with a set of indicative devices and browsers, we underline the major good practices that developers should have under consideration when they build JL selectors. The main outcomes of this assessment are outlined below for each stakeholder.

- **Developers:** Should choose carefully which selectors to avoid in their programs according to the DOM size of the document they interact with. Even the way of usage of some selectors is important in order to avoid unwanted overhead. Good selector practices underlined above, should become part of the training in programming with JLs. Mobile application developers should prefer jQuery and ExtJS.
- **JL Communities:** Our intention was not to name the best library, but to reveal to their supporting community the drawbacks of their DOM traversal engines and help them to improve them. Surprisingly, the results of jQuery extensions selectors tests revealed that some newer versions of jQuery perform slower that older ones. JL Communities should guide developers towards the adoption of good selectors' choice.

Regarding future work, our aim with jsperf tests-suite is to produce a data set with measurements produced by all browsers that are available, running on a representative set of currently used mobile devices. The enhancement and systematic categorization of those measurements can be useful for further analysis for JL communities.

The first version of Selectors Classification tool helps developers, by just uploading the output produced by dynatrace, to quickly categorize the selectors of their programs. This tool is open source and we plan to integrate it as browser plugin in a future version. Future enhancements could be online suggestions on optimize selectors. Moreover, we plan to build a similar framework for evaluating the performance of DOM traversal and manipulation functions of jQuery.

REFERENCES

Aberdeen Group, First Class Mobile Application Performance management, Research brief, Aug. 2012.

Harris Interactive (2013). 3 Seconds or Else: Survey Shows Mobile Performance a Make or Break for Holiday Sales, Online survey conducted within the U.S. by Harris Interactive on behalf of Compuware APM from October 14–16, 2013. www.compuware.com/content/dam/compuware/apm/assets/pdfs/Compuware%20APM%202013%20Holiday%20Survey%20Report.pdf

Google I/O Keynote, May 15-17, 2013 Moscone Center, San Francisco.

Richards G., Lebresne S., Burg B. and Vitek J., 2010. An analysis of the dynamic behavior of JavaScript programs. ACM SIGPLAN conf. on Programming language design and implementation (PLDI). Canada, Toronto, pp 1-12.

Rosales-Morales V.Y., Alor-Hernández G., Juárez-Martínez U., 2011. An overview of multimedia support into JavaScript-based Frameworks for developing RIAs. In U. Electrical Communications and Computers (CONIELECOMP), 21st Int. IEEE Conference, San Andres Cholula pp. 66 - 70.

Westermann D., Happe J, Zdrahal P, Moser M, and Reussner R. 2013. Performance-Aware design of web application front-ends. In Proc. of 13th conf. on Web Engineering (ICWE'13), Springer-Verlag, Berlin, Heidelberg, pp132-139.