A comparative evaluation of JavaScript execution behavior

Martinsen, Jan Kasper; Grahn, Håkan; Isberg, Anders

Published in:
Web Engineering

DOI:
10.1007/978-3-642-22233-7_35

2011

A Comparative Evaluation of JavaScript Execution Behavior

Jan Kasper Martinsen Håkan Grahn
Blekinge Institute of Technology, Karlskrona, Sweden
{jan.kasper.martinsen,hakan.grahn}bth.se

Anders Isberg
Sony Ericsson Mobile Communications AB, Lund, Sweden
Anders.Isberg@sonyericsson.com

Abstract

JavaScript is a dynamically typed and object-based scripting language with runtime evaluation. It has emerged as an important language for client-side computation of web applications. Previous studies have shown differences in behavior between established JavaScript benchmarks and real-world web applications. However, there still remains several important aspects to explore.

In this paper, we compare the JavaScript execution behavior for four application classes, i.e., four established JavaScript benchmark suites, the start pages for the first 100 sites on the Alexa top list, 22 different use cases for Facebook, Twitter, and Blogger, and finally, demo applications for the emerging HTML5 standard. Our results extend previous studies by identifying the importance of anonymous functions, showing that just-in-time compilation often decreases the performance of real-world web applications, a more thorough and detailed analysis of the use of the `eval` function, and a detailed instruction mix evaluation.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level; CR-number2 [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

The World Wide Web has become an important platform for many applications and application domains, e.g., social networking, electronic commerce, on-line libraries, and map services. These type of applications are often collectively referred to as web applications [36]. Web applications [36] can be defined in different ways, e.g., as an application that is accessed over the network from a web browser, as a complete application that is solely executed in a web browser, and of course various combinations thereof. Social networking web applications, such as Facebook [27], Twitter [23], and Blogspot [29] have turned out to be immensely popular, being within the top-25 web sites on the Alexa list [4] of most popular web sites. All three use the interpreted language JavaScript extensively for their implementation, and as a mechanism to improve both the user interface and the interactivity.

JavaScript [20, 13] was introduced in 1995 as a way to introduce dynamic functionality on web pages that were executed on the client side. JavaScript has reached widespread use through its ease of deployment and the increasing popularity of certain Web Applications [32]. For example, we have found that nearly all of the first 100 entries in the Alexia-top sites list have some sort of JavaScript functionality embedded. JavaScript is a dynamically typed, object-based scripting language with run-time evaluation. The execution of a JavaScript program is done in a JavaScript engine [17, 38, 26], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. The performance of the JavaScript engine is important in order to develop and employ powerful new web applications, and different browser vendors constantly try to outperform each other.

In order to evaluate the performance of JavaScript engines, several benchmark suites have been proposed. The most well-known are Dromaeo [9], V8 [16], SunSpider [37], and JSBenchmark [22]. However, two previous studies have pointed out that the execution behavior of existing JavaScript benchmarks differs in several important aspects [30, 31].

In this study we compare the execution behavior of four different application classes, i.e., (i) four established JavaScript benchmark suites, (ii) the start pages for the first 100 sites on the Alexa top list, (iii) 22 different use cases for Facebook, Twitter, and Blogger, and finally, (iv) demo applications for the emerging HTML5 standard. Our measurements are performed on WebKit [38], one of the most commonly used browser environments in mobile terminals.

We extend previous studies [30, 31] with several important contributions.

- First, we extend the execution behavior analysis with two new application classes, i.e., reproducible use cases of social network applications and HTML5 applications.
- Second, we identify the importance of anonymous functions. We have found that anonymous functions [8] are used much more frequently in real-world web applications than in the existing JavaScript benchmark suites.
- Third, our results clearly show that just-in-time compilation often *decreases* the performance of real-world web applications, while it increases the performance for most of the benchmark applications.
- Fourth, a more thorough and detailed analysis of the use of the `eval` function.
- Fifth, we provide a detailed instruction mix measurement, evaluation, and analysis.

The rest of the paper is organized as follows; In Section 2 we introduce JavaScript and JavaScript engines along with the most important related work. Section 3 presents our experimental methology, while Section 4 presents the different application classes that

we evaluate. Our experimental results are presented in Section 5. Finally, we conclude our findings in Section 6.

2. Background and related work

2.1 JavaScript

An important trend in application development is that more and more applications are moved to the World Wide Web [34]. There are several reasons for this, e.g., accessibility and mobility. These applications are commonly known as web applications [36]. Popular examples of such applications are: Webmails, online retail sales, online auctions, wikis, and many other applications. In order to develop web applications, new programming languages and techniques have emerged. One such language is JavaScript [13, 20], which has been used especially in client-side applications, i.e., in web browsers, but are also applicable in the server-side applications. An example of server-side JavaScript is node.js [28], where a scalable web server is written in JavaScript.

JavaScript [13, 20] was introduced by Netscape in 1995 as a way to allow web developers to add dynamic functionality to web pages that were executed on the client side. The purposes of the functionality were typically to validate input forms and other user interface related tasks. JavaScript has since then gained momentum, through its ease of deployment and the increasing popularity of certain web applications [32]. From the first 100 entries in the Alexa-top sites list, we have found that nearly all of them had some sort of JavaScript functionality embedded.

JavaScript is a dynamically typed, prototype, object-based scripting language with run-time evaluation. The execution of a JavaScript program is done in a JavaScript engine [17, 26, 38], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. Due to the popularity of the language, there have been multiple approaches to increase the performance of the JavaScript engines, through well-known optimization techniques such as JIT related techniques, fast property access, and efficient garbage collections [14, 15].

The execution of JavaScript code is often invoked in web application through events. Events are JavaScript functionalities that are executed at certain occasions, e.g., when a web application has completed loading all of its elements, when a user clicks on a button, or events that executes JavaScript at certain regular time intervals. The last type of event is often used for so-called AJAX technologies [3]. Such AJAX requests often transmit JavaScript code that later will be executed on the client side, and can be used to automatically update the web applications. Another interesting property of JavaScript within web applications, is that there is no mechanism like hardware interrupts. This means that the web browser usually "locks" itself while waiting for the JavaScript code to complete its execution, e.g., a large loop-like structure, which may degrade the user experience. Partial solutions exist, e.g., in Chrome where each tab is an own process, and a similar solution exists in WebKit 2.0 [1].

2.2 Related work

With the increasing popularity of web applications, it has been suggested that the web browser could serve as a general platform for applications in the future. This would imply that JavaScript needs increased performance. Further, it also mean that one would need to look deeper into the workload of actual web applications. This process is in its early phases, but there are several examples of interesting work [27, 5]. Two concurrent studies [30, 31] explicitly compare the JavaScript execution behavior of web applications as compared to existing JavaScript benchmark suites.

The study by Ratanaworabhan et al. [30] is one of the first studies that compares JavaScript benchmarks with real-world web applications. They instrumented the Internet Explorer 8 JavaScript runtime in order to get their measurements. Their measurements were focused on two areas of the JavaScript execution behavior, i.e., (i) functions and code, and (ii) events and handlers. Based on the results, they conclude that existing JavaScript benchmarks are not representative of many real-world web applications and that conclusions from benchmark measurements can be misleading. Examples of important differences include different code sizes, web applications are often event-driven, no clear hotspot function in the web applications, and that many functions are short-lived in web applications. They also studied memory allocation and object lifetimes in their study.

The study by Richards et al. [31] also compares the execution behavior of JavaScript benchmarks with real-world web applications. In their study, they focus on the dynamic behavior and how different dynamic features are used. Examples of dynamic features evaluated are prototype hierarchy, the use of `eval`, program size, object properties, and hot loop (hotspots). They conclude that the behavior of existing JavaScript benchmarks differ on several of these issues from the behavior of real web applications.

3. Experimental methodology

3.1 Experimental procedure

In this paper we have performed the following experiments: A set of html5 demos, a set of use-cases of a number of webpages. From these we have extracted how certain JavaScript functions are used, how much time is spent on JavaScript execution and record which opcodes that are interpreted.

3.2 Experimental environment

The measurements are made on a modified version of GTK branch of webkit (r69918) and a modified version of Mozilla Firefox with FireBug JavaScript profiler. The modified versions are such that downloaded data are stored locally, so that when an operation is repeated we reload data from local storage. When JavaScript code is executed, we have enabled such that the execution is timed (in milliseconds) and that interpreted bytecodes are recorded. In addition we have compiled two versions, one where JIT compilation is enables and one where JIT compilation is disabled. To perform experiments that require user interaction, we have instrumented the Autoit scripting tool to perform a set of cases.

All the experiments are run on a centrino duo laptop with 2GB of memory with a Windows Vista, and and Ubuntu 10.04 running as a virtual image ontop of this system. This setup was nessessary as we found no altenatives with the equvialent number of features as Autoit on Ubuntu.

4. Application classes

An important issue to address when executing JavaScript applications is to obtain reproducable results, especially since the JavaScript code may change between reloads of the same url address. We have addressed this by downloading the JavaScript code locally, and run the code locally. Further, in most cases we also execute the code several times, e.g., up to ten times in the just-in-time compilation comparison in Section 5.1 and then take the best execution time for each case.

4.1 JavaScript benchmarks

There exist a number of established JavaScript benchmark suites, and in this study we use the four most known: Dromaeo [25],

[1] http://www.techradar.com/news/software/webkit-2-0-announced-taking-leaf-from-chrome-682414

V8 [16], Sunspider [37], and JSBenchmark [22]. The applications in these benchmark suites generally fall into two different categories: (i) testing of a specific functionality, e.g., string manipulation or bit operations, and (ii) ports of already existing benchmarks that are used extensively for other programming environments [2].

For instance, among the V8 benchmarks are the benchmarks Raytrace, Richards, Deltablue, and Earley-Boyer. Raytrace is a well-known computational extensive graphical algorithm that is suitable for rendering scenes with reflection. The overall idea is that for each pixel in the resulting image, we cast a ray through a scene and the ray returns the color of that pixel based on which scene objects each ray intersects [35].

Richards simulates an operating system task dispatcher, Deltablue is a constraint solver, and Earley-Boyer is a classic scheme type theorem prover benchmark. However, the Dromaeo benchmarks do test specific features of the JavaScript language and is in this sense more focused on specific JavaScript features.

Typical for the established benchmarks is that they often are problem oriented, meaning that the purpose of the benchmark is to accept a problem input, solve this certain problem, and then end the computation. This eases the measurement and gives the developer full control over the benchmarks, and increases the repeatability.

4.2 Web applications - Alexa top 100

The critical issue in this type of study is which web applications that can be considered as representative. Due to the distributed nature of the Internet, knowing which web applications are popular is difficult. Alexa [4] offers software that can be installed in the users' web browser. This software records which web applications are visited and reports this back to a global database. From this database, a list over the most visited web pages can be extracted. In Table 2 we present the 100 most visited sites from the Alexa list. In our comparative evaluation, we have used the start page for each of these 100 most visited sites as representatives for popular web applications.

In addition to evaluating the JavaScript performance and execution behavior of the first page on the Alexa top-list, we have created use cases where we measure the JavaScript performance of a set of social networking web applications. These use cases are described in the next section.

4.3 Web applications - Social network use cases

There exists many so-called social networking web applications [39], where Facebook [27] is the most popular one [4, 11]. There are even examples of countries where half of the population use Facebook to some extent during the week [10]. The users of a social networking web application can locate and keep track of friends or people that share the same interests. This set of friends represents each user's private network, and to maintain and expand a user's network, a set of functionalities is defined.

In this paper we study the social networking web applications Facebook [27], Twitter [23], and Blogger [6]. In a sense, Facebook is a general purpose social networking web application, with a wide range of different functionalities. Further, Facebook also seems to have the largest number of users.

Twitter [23] is for writing small messages, so called "tweets", which are restricted to 160 characters (giving a clear association to SMS). The users of Twitter are able to follow other people's tweets, and for instance add comments in form of twitts to their posts.

BlogSpot is a blogging web applications, that allows user to share their opinion wide range of people through writing. The writing (a so-called blog post) might read, and the person that reads this, can often add an comments to the blog post.

BlogSpot [6].

While the benchmarks have a clear purpose, with a clearly defined start and end state, social networking web applications behave more like operating system applications, where the user can perform a selected number of tasks. However, as long as the web application is viewed by the user, it often remains active, and (e.g., Facebook) performs a set of underlying tasks.

To make a characterization and comparison easier, we have defined a set of use cases, with clear start and end states. These use cases are intended to simulate common operations and to provide repeatability of the measurements. The use cases represent common user behavior in Facebook, Twitter, and BlogSpot. They are based on personal experience, since we have not been able to find any detailed studies of common case usage for social networks. The use cases are designed to mimic user behavior rather than exhausting JavaScript execution.

Figure 1, 2, and 3 give an overview of the different use cases that we have defined for Facebook, Twitter, and BlogSpot, respectively. Common for all use cases are that they start with the user login. From here the user has multiple options.

For Facebook, the user first logs in on the system. Then, the user searches for an old friend. When the user finds this old friend, the user marks him as a "friend", an operation where the user needs to ask for confirmation from the friend to make sure that he actually is the same person. This operation is a typical example of an use case, which in turn is composed of several sub use cases: 0 `-login/home`, 0.3 `-find friend`, 0.3.1 `-add friend`, and 0.3.1.0 `-send request`, as shown in Figure 1.

All use cases start with the login case, and we recognize an individual operation, such as 0.3.1 `-add friend` as a sub use case, though it must complete previous use cases. Further, we do allow use cases that goes back and forth between use cases. For example in Figure 2, if we want to both choose the option 0.1.0 `-follow` and 0.1.1 `-mention`, then we would need to visit the following sub use cases: 0 `-login/home`, 0.1 `-find person`, 0.1.0 `-follow`, 0.1 `-find person`, and 0.1.1 `-mention`.

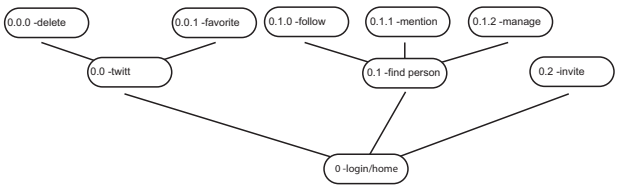

Figure 2. Use cases to characterize the JavaScript workload of Twitter.

To enhance repeatability, we use the AutoIt scripting environment [7] to automatically execute the various use cases in a controlled fashion. As a result, we can make sure that we spend the same amount of time on the same or similar operations, such as to type in a password or click on certain buttons. This is suitable for the selected use cases.

4.4 HTML5 and the canvas element

There have been several attempts to add more extensive interactive multimedia to web applications. These attempts could be roughly

Table 1. A summary of the benchmark suites used in this paper.

Benchmark suite	Applications
Dromaeo [9]	3d-cube, core-eval, object-array, object-regexp, object-string, string-base64
V8 [16]	crypto, deltablue, earley-boyer, raytrace, richards
SunSpider [37]	3d-morph, 3d-raytrace access-binary-trees, access-fannkuch, access-nbody, access-nsieve bitops-3bit-bits-in-byte, bitops-bits-in-byte, bitops-bitwise-and, bitops-nsieve-bits controlflow-recursive crypto-aes, crypto-md5, crypto-sha1 date-format-tofte, date-format-xparb math-cordic, math-partial-sums, math-spectral-norm regexp-dna string-fasta, string-tagcloud, string-unpack-code, string-validate-input
JSBenchmark [22]	Quicksort, Factorials, Conway, Ribosome, MD5, Primes, Genetic Salesman, Arrays, Dates, Exceptions

Table 2. A summary of the 100 most visited sites in the Alexa top-sites list [4] used in this paper (listed alfabetically).

163.com	1e100.net	4shared.com	about.com	adobe.com	amazon.com	ameblo.jp
aol.com	apple.com	ask.com	baidu.com	bbc.co.uk	bing.com	blogger.com
bp.blogspot.com	cnet.com	cnn.com	conduit.com	craigslist.org	dailymotion.com	deviantart.com
digg.com	doubleclick.com	ebay.com	ebay.de	espn.go.com	facebook.com	fc2.com
files.wordpress.com	flickr.com	globo.com	go.com	google.ca	google.cn	google.co.id
google.co.in	google.co.jp	google.co.uk	google.com	google.com.au	google.com.br	google.com.mx
google.com.tr	google.de	google.es	google.fr	google.it	google.pl	google.ru
hi5.com	hotfile.com	imageshack.us	imdb.com	kaixin001.com	linkedin.com	live.co
livedoor.com	livejasmin.com	livejournal.com	mail.ru	mediafire.com	megaupload.com	megavideo.com
microsoft.com	mixi.jp	mozilla.com	msn.com	myspace.com	nytimes.com	odnoklassniki.ru
orkut.co.in	orkut.com	orkut.com.br	photobucket.com	pornhub.com	qq.com	rakuten.co.jp
rapidshare.com	redtube.com	renren.com	sina.com.cn	sohu.com	soso.com	taobao.com
tianya.cn	tube8.com	tudou.com	twitter.com	uol.com.br	vkontakte.ru	wikipedia.org
wordpress.com	xhamster.com	xvideos.com	yahoo.co.jp	yahoo.com	yandex.ru	youku.com
youporn.com	youtube.com					

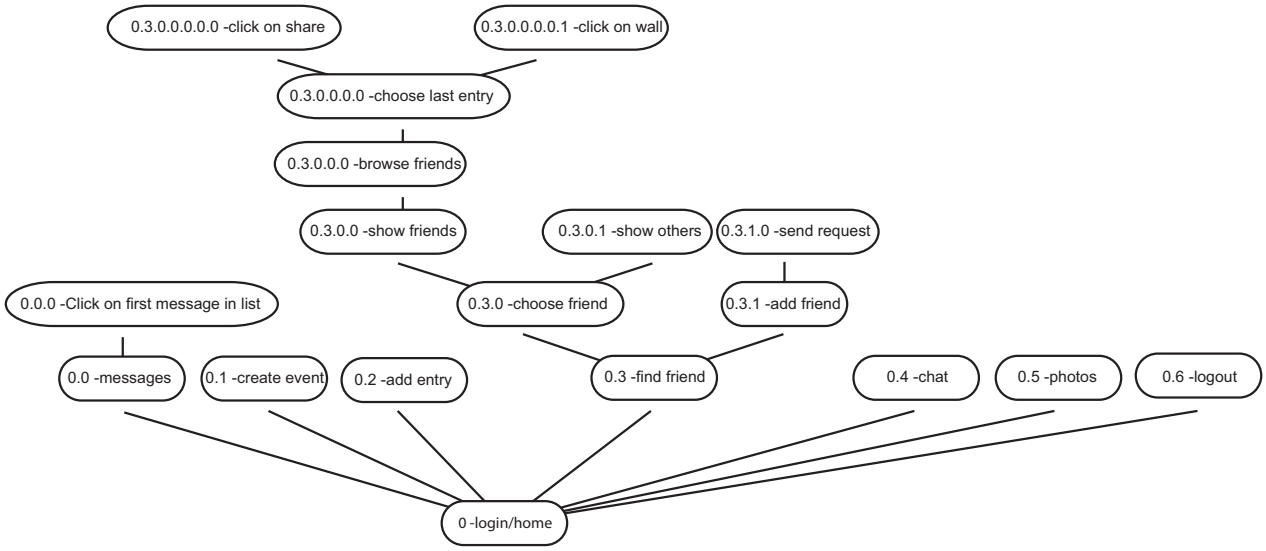

Figure 1. Use cases to characterize the JavaScript workload of Facebook.

divided into two groups: plug-in technologies and scriptable extension to web browsers. Plug-ins are programs that run on top of the web browser. The Plug-ins can execute some special type of programs, and well known examples are Adobe Flash, Java Applets, Adobe Shockwave, Alambik, Internet C++, and Silverlight. These require that the user downloads and installs a plug-in program before they can execute associated programs. Scriptable extensions introduce features in the web browser that can be manipulated through, e.g., JavaScript.

HTML5 [19] is the next standard version of the HyperText Markup Language. The Canvas in element HTML5 [18] has been agreed on by a large majority of the web browser vendors, such as Mozilla FireFox, Google Chrome, Safari, Opera and Internet Ex-

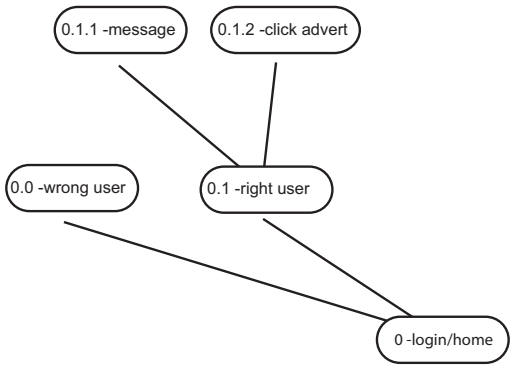

Figure 3. Use cases to characterize the JavaScript workload of BlogSpot.

plorer 9[2]. The Canvas element opened up for adding rich interactive multimedia to web application. The canvas element allows the user to add dynamic scriptable rendering of geometric shapes and bitmap images in a low level procedural manner to web applications. A similar technology, albeit at a higher level, is scalable vector graphics [24].

This element opens up for more interactive web applications. As an initiative for programmers to explore and develop the canvas element further, a series of competitions have been arranged [1, 33, 21]. The JS1k competition got 460 entries. The premise for this competition was that the entries should be less than 1024 bytes in total (with an extra bonus if they would fit inside a tweet). Further, it was forbidden to use external elements such as images. The entries vary in functionality and features, which can be illustrated by the top 10 entries, shown in Table 3, where half of them are something else than a game.

Table 3. The top-10 contributions in the JS1K competition.

	Name	Developer
1	Legend Of The Bouncing Beholder	@marijnjh
2	Tiny chess	Oscar Toledo G.
3	Tetris with sound	@sjoerd_visscher
4	WOLF1K and the rainbow characters	@p01
5	Binary clock (tweetable)	@alexeym
6	Mother fucking lasers	@evilhackerdude
7	Graphical layout engine	Lars Ronnback
8	Crazy multiplayer 2-sided Pong	@feiss
9	Morse code generator	@chrissmoak
10	Pulsing 3d wires	@unconed

[2] However is unclear whenever it will be supported in the final version of Internet Explorer 9.

5. Experimental results

5.1 Comparison of the effect of just-in-time compilation

We have compared the execution time for WebKit where just-in-time compilation(JIT) has been enabled, against the execution time where the JIT compiler has been disabled (NOJIT). When JIT has been disabled the JavaScript is interpreted as bytecode. All modifications are made to the JavaScriptCore engine, and we have used the GTK branch of the WebKit source distribution (r69918).

We have divided the execution time of a JIT version (JIT) with the execution time of the interpretive mode (NOJIT), i.e., $T_{exe}(JIT)/T_{exe}(NOJIT)$. That means, if

$$T_{exe}(JIT)/T_{exe}(NOJIT) \geq 1$$

then the JavaScript program runs slower when just-in-time compitation is enabled. We have measured the execution time that each method call uses in the JavaScriptCore in WebKit.

In Figure 4, Figure 5, and Figure 6 we have plotted the values of $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Alexa top-100 web sites, a number of use cases for social network applications, and the first 109 JS1K demos, respectively. We have plotted Figure 4 and Figure 6 with a logarithmic scale on the y-axis, enabling us to make comments about how effective JIT is when it is successful. Due to a small number of entries and variation in the data, we have plotted the results in Figure 5 in a linear scale on the y-axis (in contrast to Figures 4 and 6). We have sorted the results such that the least successful are placed to the left.

We have measured the workload of the first 100 web applications on the Alexa list, without supplying these with any kind of interaction. We have measured the first 109 JavaScript demos from the JS1K competition that had a strong focus on the canvas element from html5 without any interaction, even though some of them suggested interaction (such as computer games).

We have also measured a set of web applications that could be loosely described as social networks (`facebook.com`, `twitter.com` and `blogsplot.com`). Each of these are among the first 100 entries in the Alexa list. For these we have defined a set of use-cases. The use-cases presented in Figure 5 are extension of each other, as discussed in Section 4. For instance, case0 is extended into case1, and case1 is then extended into case2. Further, we have evaluated the effect of just-in-time compilation also on four benchmark suites, i.e., Dromaeo, V8, Sunspider, and JSBenchmark.

Each application, both for web applications or benchmarks, is executed 10 times each, where the best one out of the 10 executions is selected for comparison. For the web applications we record queries with a proxy server to minimize the chance that the JavaScript code changes between each time.

In Figure 4 we see that for 58 out of the 100 web applications, JIT compilation actually *increases* the execution time. Two of the web applications, `googleusercontent.com` and `bp.blogspot.com`, were both unavailable at the time of the experiment[3]. We see that even though more than half of the web applications had a prolonged execution time when using JIT compilation, those applications that did benefit from JIT compilation did improve their execution time significantly. For example, for `craiglist.com` JIT improved the execution time with a factor of 5000. For the search engine `yahoo.co.jp` JIT did *increase* the execution time by a factor of 3.99.

In Figure 6 we see that JIT compilation did *increase* the execution time for 59 out of the 109 JS1K demos. When JIT fails, it increases the execution time by a factor of up to 75.02 times. When JIT is successful, it decreases the execution time by up to a factor of 263.

[3] `blogspot.com` was available as we see in later experiments, but Alexa specified `bp.blogspot.com`

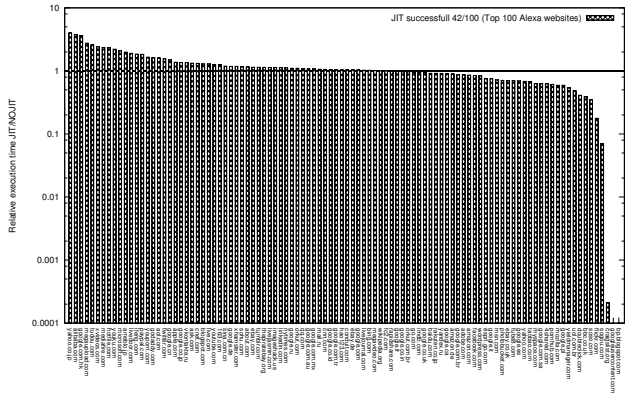

Figure 4. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Alexa top 100 web sites.

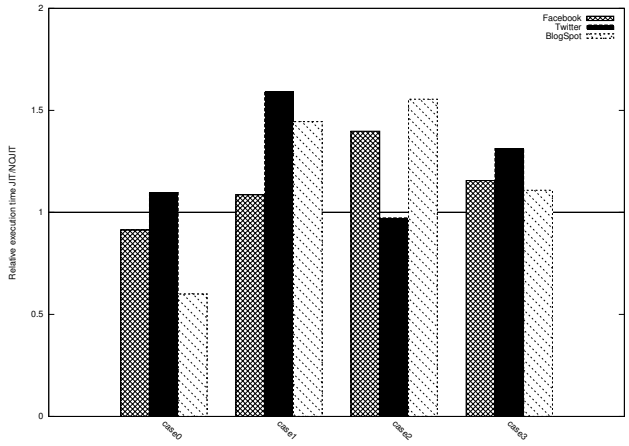

Figure 5. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for 4 use cases from three different social network applications.

In Figure 7 we have evaluated 4 out of the 5 V8 benchmarks. The last benchmark, earley-boyer, did not execute correctly with the selected version of WebKit. We see that JIT compilation is successful in 3 out of 4 cases. We see that best improvement is a factor of 1.9, while in the worst case the execution time is increased by a factor of 1.14.

In Figure 8 we see that JIT compilation improves the execution time for 3 out of 6 applications in the Dromaeo benchmark suite. The largest improvement is by a factor of 1.54, while in the case where JIT performs worst the execution time is is increased by a factor of 1.32.

In Figure 9, where the results for the SunSpider benchmark suite is presented, all the entries run equally fast or faster when JIT compliation is enabled. The largest improvement is by a factor of 16.4 for the `string-validate-input` application, and the smallest improvement is 1.0, i.e., none, for the `date-format-tofte` application.

In Figure 10 we see that JIT compilation successfully decreases the execution time for 7 out of 10 applications in the JSBenchmark suite. The largest decrease in execution time is by a factor of 1.6. The largest decrease in the execution time is by a factor of 1.07.

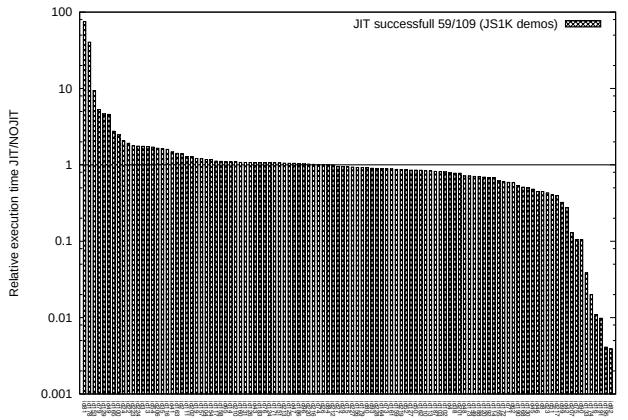

Figure 6. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the first 109 JS1K demos.

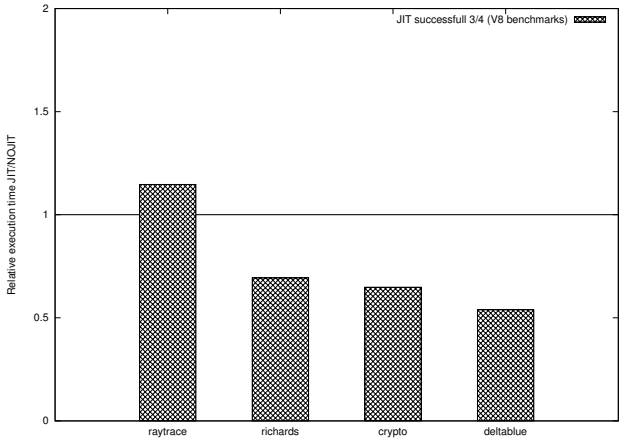

Figure 7. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the V8 benchmarks.

In summary, we can conclude that JIT compilation decreases the execution time for most the benchmarks. However, for the web applications JIT compilation actually *increases* the execution time for more than half of the studied web applications. In the worst case, we found that the execution time was prolonged by up to 75 times in the worst case (`id81` in the JS1K demos).

5.2 Comparison of bytecode instruction usage

We have recorded the number of executed bytecode instructions in the JavaScriptCore for all the benchmarks and for the first 100 entries in the Alexa top list. We do present the results of the Alexa top 100 applications versus the SunSpider benchmark, since these two application sets differed most.

The SunSpider benchmark uses a smaller subset of the available instructions than the Alexa websites do. The Alexa websites use 118 out of 139 instructions while the SunSpider benchmarks only use 82 out of the 139 available bytecode instructions (this includes instructions that are used mostly for debugging purposes in WebKit). We have grouped the instructions loosely based on instructions that have similar behavoir. The instruction groups are: arith-

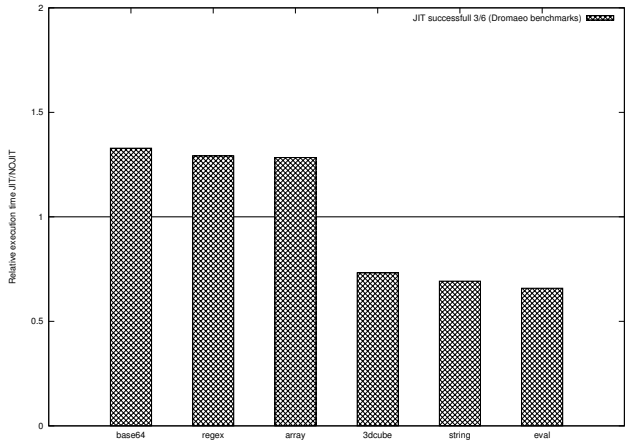

Figure 8. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Dromaeo benchmarks.

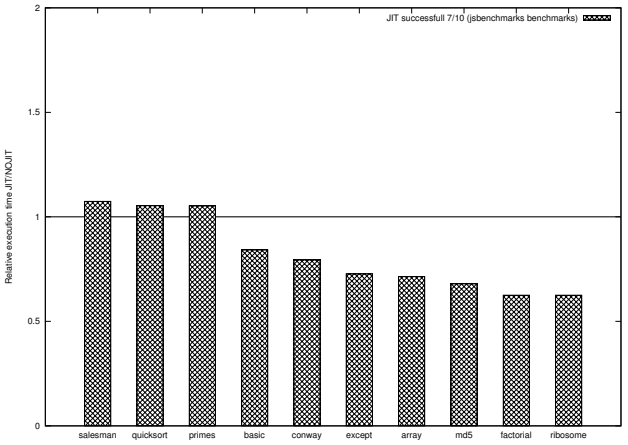

Figure 10. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the JSBenchmark.

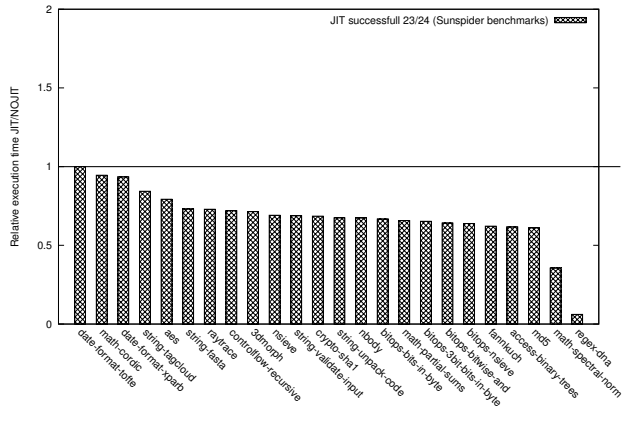

Figure 9. Relative execution time $T_{exe}(JIT)$ / $T_{exe}(NOJIT)$ for the Sunspider benchmarks.

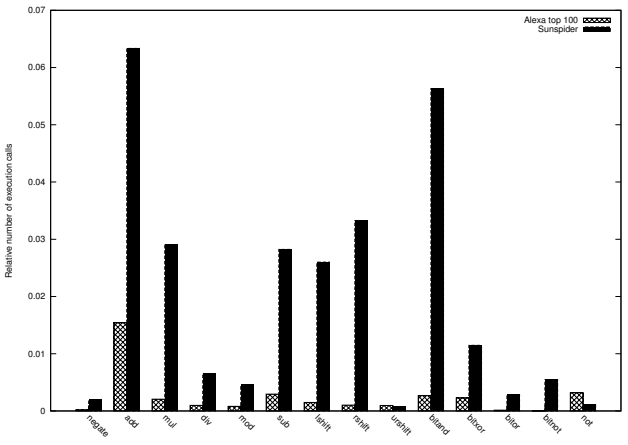

Figure 11. Aritmetic instructions for Alexa websites and Sunspider benchmarks

metic/logical, prototype and object manipulation, and branches and jumps.

In Figure 11 we see that arithmetic/logical instructions are much more intensively used in the SunSpider benchmarks than in the web applications covered by Alexa top 100. We also see that the SunSpider benchmarks often use bit operations (such as left and right shift) which are very rarely used in the websites. This observation suggests that even though most of these operations are important and well known in programming languages such as C, it seems like these are rarely used in web applications. JavaScript moves away from hardware, making little use of bit oriented operations. The only operation that seems to be used more in web applications than in the benchmarks is the `not` instruction, which could be used in, e.g., comparisons.

We notice that Alexa top 100 web applications seem to use the object model of JavaScript, and therefore use the object special features more extensively than the benchmarks. In Figure 12 we see that instructions such as `get_by_id`, `get_by_id_self`, and `get_by_id_proto` are used much more in the web applications than in the benchmarks. Features such as classless prototyped programming, are usually associated with research oriented programming languages, and it is likely that these concepts are not well reflected in a set of benchmarks that have been ported from a traditional programming language. A closer inspections of the source code of the benchmarks confirms this. It seems like many of the benchmarks are embedded into typically object-based constructions, which assist in measuring execution time and other benchmarks related tasks. However, these object-based constructions are only rarely a part of the compute intensive parts of the benchmark.

The observation above is further supported in Figure 13, by looking at instructions such as `get_val` and `put_val`, which the SunSpider benchmarks use more extensively than the web applications. This suggests that the benchmarks do not take advantage of the JavaScript feature of classless prototype features, and rather tries to emulate the data structures in the original benchmarks which they often were ported from.

For the branch and jump bytecode instruction group, we observe in Figure 14 that jumps related to objects are common in Alexa, while jumps that are typically assosiated with conditional statements, such as loops are much more used in the benchmarks. A larger number of `jmp` instructions also illustrates the importance of function calls in web applications.

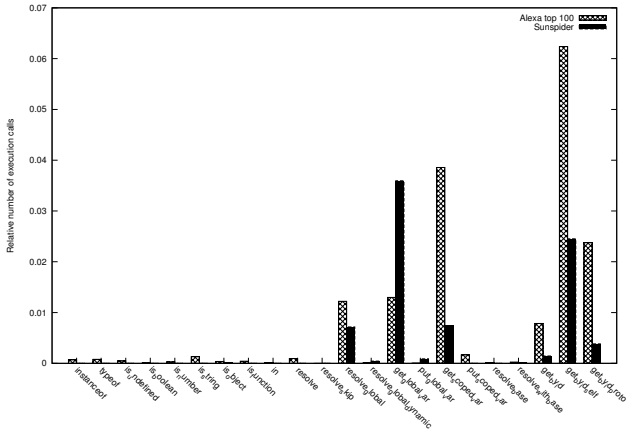

Figure 12. Prototype related instructions for the Alexa top 100 websites and the SunSpider benchmarks.

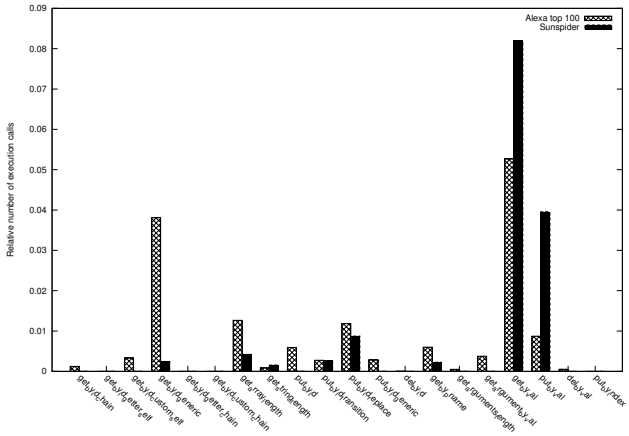

Figure 13. Prototype related instructions for the Alexa top 100 websites and the SunSpider benchmarks.

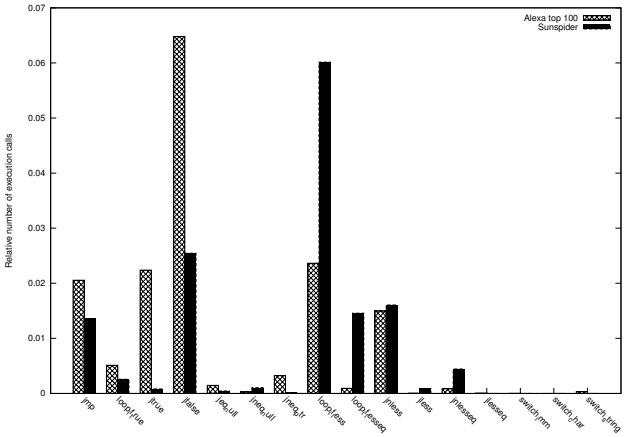

Figure 14. Branch and jump related instructions for the Alexa top 100 websites and the SunSpider benchmarks.

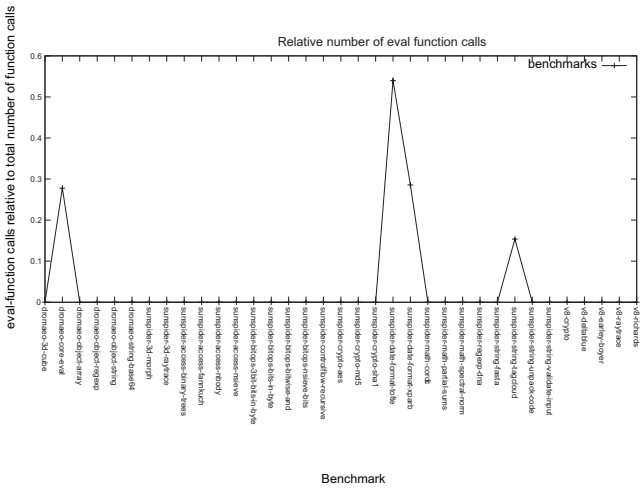

Figure 15. Number of `eval` calls relative to the number of total function calls in the Dromaeo, V8, and SunSpider benchmarks.

5.3 Usage of the `eval` function

One feature of JavaScript is that it uses evaluate (`eval`) function calls, that executes a given string of JavaScript source code at runtime. To extract information of how frequently `eval` calls are executed, we have used the FireBug [12] JavaScript profiler to extract this information. We have then measured the number of `eval` calls relative to the total number of function calls ($No.of$ `eval`$calls/Totalno.of\,functioncalls$).

In Figure 15 we see that such functions are rather rare in the benchmarks, apart from three instances. For the benchmarks, we observe that `eval` is used in only 4 out of 35 benchmarks. However, they use it quite extensively. The `dromaeo-core-eval` benchmark has 0.27, `sunspider-date-format-tofte` has 0.54, `sunspider-date-format-xparb` has 0.28, and `sunspider-string-tagcloud` has 0.15 relative number of `eval` calls. This accounts for an average relative number of `eval` calls of 0.31 for these four benchmarks. From their name (e.g., `eval-test` in the Dromaeo benchmark), by inspection of the JavaScript code and the amount of `eval` calls, we suspect that these benchmarks were designed to test the eval function.

For the Alexa top sites list, we see in Figure 16 that the `eval` function is used more frequently. 44 out of 100 sites use the `eval` function. On average, the relative number of `eval` calls is 0.11. However, we see in the figure that there are web applications with a large relative number of `eval` calls, such as `sina.com.cn` where 55% of all function calls are `eval` calls.

5.4 Anonymous function calls

An anonymous function call is a call to a function that does not have a name. In many programming languages this is not possible, but it is possible to create such functions in JavaScript. Since this programming construct is allowed in JavaScript, we would like to find out how common it is in JavaScript benchmarks and web applications

By inspection, we found that 3 of the anonymous function calls in the benchmarks were instrumentation of the benchmark to measure execution time. We have measured the number of anonymous function calls with the FireBug JavaScript profiler. If we removed these 3 function calls we found that 17 out of the 35 benchmark use anonymous function calls (to a variable degree). For the entries

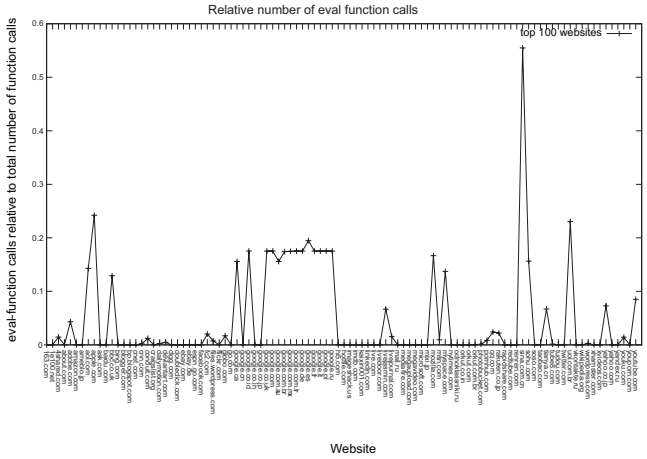

Figure 16. Number of `eval` calls relative to the number of total function calls for the first 100 entries in the Alexa list.

in the top 100 Alexa websites, we found that 74 out of 100 sites use anonymous function calls. The relative number of anonymous function calls in the benchmarks and the Alexa top 100 sites are shown in Figure 17.

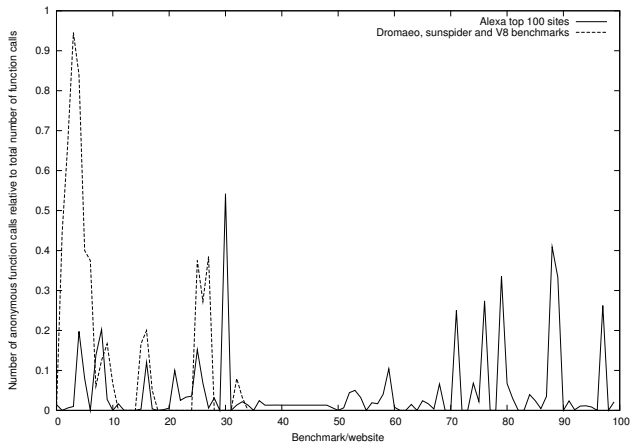

Figure 17. Use of anonymous function calls.

We see that certain of the benchmarks use anonymous function calls more extensively. Some of these are tailored to specifically test the use of anonymous function calls, much like certain benchmark were tailored to test eval in Section 5.3.

6. Conclusions

In this paper we have evaluated the execution behavior of javaScript for four different application classes, i.e., four JavaScript benchmark suites, popular web sites, use cases from social networking applications, and the emerging HTML5 standard. The measurements have been performed in the WebKit browser and JavaScript execution environment.

We have found that the behavior of benchmarks and real-world web applications differ in several significant ways:

- Just-in-time compilation is beneficial for most of the benchmarks, but actually *increases* the execution time for more than half of the web applications.
- Arithmetic byte code instructions are significantly more common in benchmarks, while prototype related instructions and branches are more common in real-world applications.
- The `eval` function is much more commonly used in web applications than in benchmark applications.
- We found that approximately half of the benchmarks used anonymous functions, while approximately 75% of the web applications used anonymous functions.

Based on the findings above, in combinations with findings in previous studies [30, 31], one can conclude that the existing benchmark suites do not reflect the execution behavior of real-world web applications.

Acknowledgments

This work was partly funded by the Industrial Excellence Center EASE - Embedded Applications Software Engineering, (`http://ease.cs.lth.se`).

References

[1] 10KApart. Inspire the web with just 10k, 2010. `http://10k.aneventapart.com/`.

[2] Ole Agesen. GC points in a threaded environment. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 1998.

[3] Therese J. Albert, Kai Qian, and Xiang Fu. Race condition in ajax-based web application. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference on XX*, pages 390–393, New York, NY, USA, 2008. ACM.

[4] Alexa. Top 500 sites on the web, 2010. http://www.alexa.com/topsites.

[5] Anneliese A. Andrews, Jeff Offutt, Curtis Dyreson, Christopher J. Mallery, Kshamta Jerath, and Roger Alexander. Scalability issues with using fsmweb to test web applications. *Inf. Softw. Technol.*, 52(1):52–66, 2010.

[6] Blogger: Create your free blog, 2010. `http://www.blogger.com/`.

[7] Jason Brand and Jeff Balvanz. Automation is a breeze with autoit. In *SIGUCCS '05: Proceedings of the 33rd annual ACM SIGUCCS conference on User services*, pages 12–15, New York, NY, USA, 2005. ACM.

[8] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *PLDI '09: Proc. of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 50–62, New York, NY, USA, 2009. ACM.

[9] Dromaeo. Dromaeo: JavaScript performance testing, 2010. http://dromaeo.com/.

[10] Eric Eldon. Facebook used by the most people within iceland, norway, canada, other cold places, 2009. http://www.insidefacebook.com/2009/09/25/faceb used-by-the-most-people-within-iceland-norway-canada-other-cold-places/.

[11] Facebook, 2010. http://www.facebook.com/press/info.php?statistics.

[12] FireBug. Firebug, javascript profiler, 2010. http://getfirebug.com.

[13] David Flanagan. *JavaScript: The Definitive Guide, 5th edition*. O'Reilly Media, 2006.

[14] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478, New York, NY, USA, 2009. ACM.

[15] Google. V8 Google JavaScript interpreter, 2008. http://code.google.com/intl/fr/api

[16] Google. V8 benchmark suite - version 5, 2010. http://v8.googlecode.com/svn/data/

[17] Google. V8 JavaScript Engine, 2010. http://code.google.com/p/v8/.

[18] Michael Grady. Functional programming using JavaScript and the HTML5 canvas element. *J. Comput. Small Coll.*, 26:97–105, December 2010.

[19] W3C HTML Working Group, 2010. `http://www.w3.org/html/wg/`.

[20] JavaScript, 2010. http://en.wikipedia.org/wiki/JavaScript.

[21] JS1k. This is the website for the 1k JavaScript demo contest #js1k, 2010. `http://js1k.com/home`.

[22] JSBenchmark, 2010. http://jsbenchmark.celtickane.com/.

[23] Balachander Krishnamurthy, Phillipa Gill, and Martin Arlitt. A few chirps about twitter. In *WOSP '08: Proceedings of the first workshop on Online social networks*, pages 19–24, New York, NY, USA, 2008. ACM.

[24] Francis Molina, Brian Sweeney, Ted Willard, and André Winter. Building cross-browser interfaces for digital libraries with scalable vector graphics (svg). In *Proc. of the 7th ACM/IEEE-CS joint conference on Digital libraries*, JCDL '07, pages 494–494, New York, NY, USA, 2007. ACM.

[25] Mozilla. Dromaeo: JavaScript performance testing, 2010. http://dromaeo.com/.

[26] Mozilla. What is SpiderMonkey?, 2010. http://www.mozilla.org/js/spidermonkey/.

[27] Atif Nazir, Saqib Raza, and Chen-Nee Chuah. Unveiling Facebook: A measurement study of social network based applications. In *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, pages 43–56, New York, NY, USA, 2008. ACM.

[28] Node.js. Evented I/O for V8 JavaScript, 2010. `http://nodejs.org/`.

[29] Ulrike Pfeil, Raj Arjan, and Panayiotis Zaphiris. Age differences in online social networking - a study of user profiles and the social capital divide among teenagers and older users in myspace. *Comput. Hum. Behav.*, 25(3):643–654, 2009.

[30] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX conference on Web application development*, WebApps'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[31] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[32] Erick Schonfeld. Gmail grew 43 percent last year. aol mail and hotmail need to start worrying, 2009. http://techcrunch.com/2009/01/14/gmail-grew-43-percent-last-year-aol-mail-and-hotmail-need-to-start-worrying/.

[33] The 5K. An award for excellence in web design and production, 2002. `http://www.the5k.org/`.

[34] W3C. World Wide Web Consortium, 2010. http://www.w3c.org/.

[35] Alan Watt. *3d Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[36] Web applications, 2010. http://en.wikipedia.org/wiki/Web_application.

[37] WebKit. SunSpider JavaScript Benchmark, 2010. http://www2.webkit.org/perf/sunspider-0.9/sunspider.html.

[38] WebKit. The WebKit open source project, 2010. http://www.webkit.org/.

[39] Wikipedia. List of social networking websites, 2010. http://en.wikipedia.org/wiki/List_of_social_networking_websites.