EVALUATING FOUR ASPECTS OF JAVASCRIPT EXECUTION BEHAVIOR IN BENCHMARKS AND WEB APPLICATIONS

Jan Kasper Martinsen, Håkan Grahn, Anders Isberg
Evaluating Four Aspects of JavaScript Execution Behavior in Benchmarks and Web Applications*

Jan Kasper Martinsen, Häkan Grahn, and Anders Isberg

1 Blekinge Institute of Technology, Karlskrona, Sweden, {jan.kasper.martinsen,hakan.grahn}@bth.se
2 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 remain several important aspects to explore.

In this study, we compare the JavaScript execution behavior of four application classes, i.e., four established JavaScript benchmark suites, the first pages of the top 100 sites on the Alexa 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 and eval functions, showing that just-in-time compilation often decreases the performance of real-world web applications, and a detailed bytecode instruction mix evaluation.

1 Introduction

The World Wide Web has become an important platform for many applications and application domains, e.g., social networking and electronic commerce. These type of applications are often referred to as web applications [36]. Web applications 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 [28], Twitter [23], and Blogger [6], have turned out to be popular, being in the top-25 web sites on the Alexa list [4] of most popular web sites. All these three applications use the interpreted language JavaScript [20] extensively for their implementation, and as a mechanism to improve both the user interface and the interactivity.

JavaScript [20] 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 popularity of

*A shorter version is published in Proc. of the 11th Int’l Conf. on Web Engineering (ICWE 2011), Lecture Notes in Computer Science No. 6757, pp. 399–402, June 2011.
 certain web applications [32]. We have found that nearly all of the first 100 entries in the Alexa top sites list use JavaScript.

JavaScript [20] 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, 27], i.e., an interpreter/virtual machine that parses and executes the JavaScript program. The popularity of JavaScript increases the importance of its run-time performance, 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, e.g., Dromaeo [9], V8 [16], SunSpider [37], and JS Benchmark [22]. However, two previous studies indicate that the execution behavior of existing 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 [4], (iii) 22 different use cases for Facebook [28], Twitter [23], and Blogger [6] (sometimes referred to as BlogSpot), and finally, (iv) 109 demo applications for the emerging HTML5 standard [18]. Our measurements are performed with 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 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 bytecode 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 methodology, 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 [29], 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]. We have found that nearly all of the first 100 entries in the Alexa top sites list use some sort of JavaScript functionality.

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, 27, 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\(^3\).

2.2 Related work

With the increasing popularity of web applications, their execution behavior as well as the performance of JavaScript engines have attended an increased focus, e.g., [28, 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 are focused on two areas of the JavaScript execution behavior, i.e., (i) functions and code, and (ii) events and handlers. They conclude that existing JavaScript benchmarks are not representative of many real-world web applications and that conclusions from benchmark measurements might be misleading. 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. They conclude that the behavior of existing benchmarks differs on several of these issues from the behavior of real web applications.

3 Experimental methodology

The experimental methodology is thoroughly described in [24]. We have selected a set of 4 application classes consisting of the first page of the 100 most popular web sites, 109 HTML5 demos from the JS1K competition, 22 use cases from three popular social networks (Facebook, Twitter, and Blogger), and a set of 4 benchmarks for measurements. We have measured and evaluated two aspects: the execution time with and without just-in-time compilation, and the bytecode instruction mix for different application classes. The measurements are made on modified versions of the GTK branch of WebKit (r69918) and Mozilla Firefox with the FireBug profiler.

Web applications are highly dynamic and the JavaScript code might change from time to time. We improve the reproducibility by modifying the test environment to download and re-execute the associated JavaScript locally (if possible). For each test an initial phase is performed 10 times to reduce the chances of execution of external JavaScript code.

Another challenge is the comparison between the social networking web applications and the benchmarks, since the web applications have no clear start and end state. To address this, we defined a set of use cases based on the behavior of friends and colleagues, and from this we created instrumented executions with the Autoit tool.

We modified our test environment in order to enable or disable just-in-time compilation. During the measurements, we executed each test case and application with just-in-time compilation disabled and enabled 10 times each, and selected the best one for comparison. We used the following relative execution time metric to compare the difference between just-in-time-compilation (JIT)
Table 1. A summary of the benchmark suites used in this paper.

<table>
<thead>
<tr>
<th>Benchmark suite</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dromaeo [9]</td>
<td>3d-cube, core-eval, object-array, object-regexp, object-string, string-base64</td>
</tr>
<tr>
<td>V8 [16]</td>
<td>crypto, deltablue, earley-boyer, raytrace, richards</td>
</tr>
<tr>
<td>JSBenchmark [22]</td>
<td>Quicksort, Factorials, Conway, Ribosome, MD5, Primes, Genetic Salesman, Arrays, Dates, Exceptions</td>
</tr>
</tbody>
</table>

and no-just-in-time-compilation (NOJIT):

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

4 Application classes

An important issue to address when executing JavaScript applications is to obtain reproducible 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 execute the code several times, 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 [26], 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 [28] 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 [28], 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.
Table 2. A summary of the 100 most visited sites in the Alexa top-sites list [4] used in this paper (listed alphabetically).

<table>
<thead>
<tr>
<th>163.com</th>
<th>1e100.net</th>
<th>4shared.com</th>
<th>about.com</th>
<th>adobe.com</th>
</tr>
</thead>
<tbody>
<tr>
<td>amazon.com</td>
<td>ameblo.jp</td>
<td>aol.com</td>
<td>apple.com</td>
<td>ask.com</td>
</tr>
<tr>
<td>baidu.com</td>
<td>bbc.co.uk</td>
<td>bing.com</td>
<td>blogger.com</td>
<td>bp.blogspot.com</td>
</tr>
<tr>
<td>cnet.com</td>
<td>cnn.com</td>
<td>conduit.com</td>
<td>craigslist.org</td>
<td>dailymotion.com</td>
</tr>
<tr>
<td>deviantart.com</td>
<td>digg.com</td>
<td>doubleclick.com</td>
<td>ebay.com</td>
<td>ebay.de</td>
</tr>
<tr>
<td>espn.go.com</td>
<td>facebook.com</td>
<td>fe2.com</td>
<td>files.wordpress.com</td>
<td>flickr.com</td>
</tr>
<tr>
<td>globo.com</td>
<td>go.com</td>
<td>google.ca</td>
<td>google.cn</td>
<td>google.co.id</td>
</tr>
<tr>
<td>google.co.in</td>
<td>google.co.jp</td>
<td>google.co.uk</td>
<td>google.com</td>
<td>google.com.au</td>
</tr>
<tr>
<td>google.com.br</td>
<td>google.com.mx</td>
<td>google.com.tr</td>
<td>google.de</td>
<td>google.es</td>
</tr>
<tr>
<td>google.fr</td>
<td>google.it</td>
<td>google.pl</td>
<td>google.ru</td>
<td>hi5.com</td>
</tr>
<tr>
<td>hotfile.com</td>
<td>imageshack.us</td>
<td>imdb.com</td>
<td>kaixin001.com</td>
<td>linkedin.com</td>
</tr>
<tr>
<td>live.co</td>
<td>livedoor.com</td>
<td>livejasmin.com</td>
<td>livejournal.com</td>
<td>mail.ru</td>
</tr>
<tr>
<td>mediawire.com</td>
<td>megupload.com</td>
<td>megavideo.com</td>
<td>microsoft.com</td>
<td>mixi.jp</td>
</tr>
<tr>
<td>mozilla.com</td>
<td>msn.com</td>
<td>myspace.com</td>
<td>nytimes.com</td>
<td>odnoklassniki.ru</td>
</tr>
<tr>
<td>orkut.co.in</td>
<td>orkut.com</td>
<td>orkut.com.br</td>
<td>photobucket.com</td>
<td>pornhub.com</td>
</tr>
<tr>
<td>qq.com</td>
<td>rakuten.co.jp</td>
<td>rapidshare.com</td>
<td>redtube.com</td>
<td>reuren.com</td>
</tr>
<tr>
<td>sina.com.cn</td>
<td>sohu.com</td>
<td>sos.com</td>
<td>taobao.com</td>
<td>tianya.cn</td>
</tr>
<tr>
<td>tube8.com</td>
<td>tudou.com</td>
<td>twitter.com</td>
<td>uol.com.br</td>
<td>vkontakte.ru</td>
</tr>
<tr>
<td>wikipedia.org</td>
<td>wordpress.com</td>
<td>xhamster.com</td>
<td>xvideos.com</td>
<td>yahoo.co.jp</td>
</tr>
<tr>
<td>yahoo.com</td>
<td>yandex.ru</td>
<td>youku.com</td>
<td>youporn.com</td>
<td>youtube.com</td>
</tr>
</tbody>
</table>

Blogger is a blogging web application, 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.

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 Blogger. 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 Blogger, 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.
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 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 Explorer 9. 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 [25].

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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Developer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Legend Of The Bouncing Beholder</td>
<td>@marijnjh</td>
</tr>
<tr>
<td>Tiny chess</td>
<td>Oscar Toledo G.</td>
</tr>
<tr>
<td>Tetris with sound</td>
<td>@sjoerd_visscher</td>
</tr>
<tr>
<td>WOLF1K and the rainbow characters</td>
<td>@p01</td>
</tr>
<tr>
<td>Binary clock (tweetable)</td>
<td>@alexeym</td>
</tr>
<tr>
<td>Mother fucking lasers</td>
<td>@evilhackerdude</td>
</tr>
<tr>
<td>Graphical layout engine</td>
<td>Lars Ronnback</td>
</tr>
<tr>
<td>Crazy multiplayer 2-sided Pong</td>
<td>@feiss</td>
</tr>
<tr>
<td>Morse code generator</td>
<td>@chrissmoak</td>
</tr>
<tr>
<td>Pulsing 3d wires</td>
<td>@unconed</td>
</tr>
</tbody>
</table>

5 Experimental results

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

We have compared the execution time 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 the JIT version with the execution time of the interpretation mode, i.e., \( T_{ex}(\text{JIT}) / T_{ex}(\text{NOJIT}) \). That means, if

\[
T_{ex}(\text{JIT}) / T_{ex}(\text{NOJIT}) \geq 1
\]

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

In Figure 4 we have plotted the values of \( T_{ex}(\text{JIT}) / T_{ex}(\text{NOJIT}) \) for a number of use cases for the top 3 social network applications, i.e., Facebook, Twitter, and Blogger, for a set of use cases. The use cases presented in Figure 4 are extensions of each other, as discussed in Section 4. For instance, case0 is extended into case1, and case1 is then extended into case2. Our results show that the execution time increases in 9 out of 12 cases when JIT is enabled. This especially pronounced for the more complicated use cases. The reason is the non-repetitive behavior of the social network application use cases.
In Figure 5 we present the relative execution time $T_{exe}(JIT) / T_{exe}(NOJIT)$ for the Alexa top 100 web sites and the first 109 JS1K demos. We have measured the workload of them without any user interaction. The results in Figure 5 show that for 58 out of the 100 web applications, JIT increases the execution time. However, for those applications that benefit from JIT, their execution times are improved significantly. For instance, the execution time for craigslist.com was improved by a factor of 5000. For yahoo.co.jp JIT increased the execution time by a factor of 3.99.

Further, in Figure 5 we see that JIT increased 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. When JIT is successful, it decreases the execution time by up to a factor of 263.

Finally, we have evaluated the effect of JIT on the four benchmark suites, i.e., Dromaeo, V8, Sunspider, and JSBenchmark, as shown in Figures 6 and 7. In Figure 6, we show the results for 4 out of 5 of the V8 benchmarks, 6 of the Dromaeo benchmarks, and 10 of the JSBenchmarks. For V8, JIT is successful in 3 out of 4 cases and the best improvement is a factor of 1.9, while in the worst case the execution time is increased by a factor of 1.14. For Dromaeo JIT

---

4 Earley-boyer, did not execute correctly with the selected version of WebKit.
improves the execution time for 3 out of 6 cases. The largest improvement is by a factor of 1.54, while largest increase in execution time is by a factor of 1.32. For the JSBenchmarks, JIT decreases the execution time for 7 out of 10 cases. The largest decrease in execution time is by a factor of 1.6. The largest increase in the execution time is by a factor of 1.07.

Finally, Figure 7 shows the results for the SunSpider benchmark. All the applications in the SunSpider benchmark suite run equally fast or faster when JIT 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 summary, JIT decreases the execution time for most of the benchmarks. In contrast, JIT increases the execution time for more than half of the studied web applications. In the worst case, the execution time was prolonged by a factor of 75 (id81 in the JS1K demos).

5.2 Comparison of bytecode instruction usage

We have measured the bytecode instruction mix, i.e., the number of executed bytecode instructions for each bytecode instruction, for the selected benchmarks and for the first 100 entries in the Alexa top list. Then, a comparison between
the web applications and the SunSpider benchmarks is done, since these two differ the most.

The SunSpider benchmarks use a smaller subset of bytecode instructions than the Alexa web sites do. The Alexa web sites use 118 out of 139 bytecode instructions, while the SunSpider benchmarks only use 82 out of the 139 instructions. We have grouped the instructions based on instructions that have similar behaviors. The instruction groups are: prototype and object manipulation, branches and jumps, and arithmetic/logical.

In Figure 8 we see that arithmetic/logical instructions are more intensively used in the SunSpider benchmarks than in the web applications covered by Alexa top 100. We also observe that the SunSpider benchmarks often use bit operations (such as left and right shift) which are rarely used in the web sites. This observation suggests that even though these operations are important in low level programming languages, it seems like these are rarely used in web applications. The only arithmetic/logical operation that is more used in web applications is the not instruction, which could be used in, e.g., comparisons.

For the branch and jump bytecode instruction group, we observe in Figure 8 that jumps related to objects are common in Alexa, while jumps that are associated with conditional statements, such as loops are much more used in the
benchmarks. A large number of jmp instructions also illustrates the importance of function calls in web applications.

We notice that Alexa top 100 web applications use the object model of JavaScript, and therefore use the object special features more than the benchmarks. In Figure 9 we see that instructions such as get_by_id, get_by_id_self, and get_by_id_proto are used more in the web applications than in the benchmarks. Features such as classless prototyped programming are rarely found in traditional programming languages which the benchmarks are ported from. A closer inspections of the source code of the benchmarks confirms this. It seems like many of the benchmarks are embedded into typical object-based constructions, which assist in measuring execution time and other benchmarks related tasks. However, these object-based constructions are rarely a part of the compute intensive parts of the benchmark.

The observation above is further supported in Figure 9, 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 JavaScript classless prototype features, and instead try to simulate the data structures found in the original benchmarks.
5.3 Usage of the eval function

One JavaScript feature is the evaluate function, eval, that evaluates and executes a given string of JavaScript source code at runtime. To extract information on how frequently eval calls are executed, we have used the FireBug [12] JavaScript profiler to extract this information. We have measured the number of eval calls relative to the total number of function calls, i.e., No. of eval calls / Total no. of function calls.

Figure 10 presents the relative number of eval calls. Our results show that eval functions are rarely being used in the benchmarks, only 4 out of 35 benchmarks use the eval function. However, these four use eval 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. From their name, e.g., eval-test in the Dromaeo benchmark, and by inspection of the JavaScript code and the amount of eval calls, we suspect that these benchmarks were designed specifically to test the eval function.

We observe in Figure 11 that the eval function is used more frequently in the Alexa top 100 web sites. 44 out of 100 web sites use the eval function. In average, the relative number of eval calls is 0.11. However, there are web
sites with a large relative number of `eval` calls, e.g., in `sina.com.cn` 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. The relative number of anonymous function calls in the benchmarks and the Alexa top 100 sites are shown in Figure 12.

We found that 3 of the anonymous function calls in the benchmarks were instrumentations of the benchmark to measure execution time. If we removed these 3 function calls we found that 17 out of the 35 benchmark used anonymous function calls to some degree. For the entries in the top 100 Alexa web sites, we found that 74 out of 100 sites used anonymous function calls. Some benchmarks use anonymous function calls extensively. However, these seems to be specifically tailored for anonymous function calls, much like certain benchmarks were tailored to test `eval` in Section 5.3.
Fig. 10. Number of eval calls relative to the total number of function calls in the Dromaeo, V8, and SunSpider benchmarks.

6 Conclusions

In this study, we have evaluated and compared 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.

Our results show that 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/logical bytecode instructions are significantly more common in benchmarks, while prototype related instructions and branches are more common in real-world web applications.
− The eval function is much more commonly used in web applications than in benchmark applications.
− Approximately half of the benchmarks use anonymous functions, while approximately 75% of the web applications use anonymous functions.
Based on the findings above, in combination with findings in previous studies [30, 31], we conclude that the existing benchmark suites do not reflect the execution behavior of real-world web applications. For example, special JavaScript features such as dynamic types, \texttt{eval} functions, anonymous functions, and event-based programming, are omitted from the computational parts of the benchmarks, while these features are used extensively in web applications. A more serious implication is that optimization techniques employed in JavaScript engines today might be geared towards workloads that only exist in benchmarks.

Acknowledgments

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

References

Fig. 12. Relative number of anonymous function calls in the Alexa top 100 web sites and the benchmarks.

21. JS1k. This is the website for the 1k JavaScript demo contest #js1k, 2010. http://js1k.com/home.
EVALUATING FOUR ASPECTS OF JAVASCRIPT EXECUTION BEHAVIOR IN BENCHMARKS AND WEB APPLICATIONS

Jan Kasper Martinsen, Håkan Grahn, Anders Isberg