JavaScript Guidelines for JavaScript Programmers
A Comprehensive Guide for Performance Critical JS Programs

Gábor Lóki and Péter Gál
Department of Software Engineering, University of Szeged, Dugonics ter 13, 6720, Szeged, Hungary

Keywords: JavaScript, Guidelines, EcmaScript 262, Performance, Embedded.

Abstract: Programming guidelines are used for almost every programming language. Guidelines can differ for each project and each programmer. In general, however they usually try to give a common format for the given project in some aspect. This aspect can be code style related or even performance related. A performance guideline tries to help programmers formulate such code which can be executed quickly by the computer. For statically compiled languages, numerous performance guidelines are available. In the web era, the JavaScript language is used extensively by many developers. For this language, the performance guidelines are not that widespread, although there are a few research papers about them. Additionally, the language has incorporated new constructs in its newer versions. In this paper, some of the new ECMAScript 6 constructs are investigated to determine if they should be used in a performance sensitive JavaScript application. The elements are compared with the ECMAScript 5.1 variants. To give a more usable set of guidelines, the tests are performed on multiple JavaScript engines ranging from server side JS engines to engines which can be used in embedded systems.

1 INTRODUCTION

Without doubt, scripting languages have been becoming increasingly popular over the last decade. The main code hosting services and developer discussion forums are publishing the results of surveys and statistics every year including the popularity of programming languages (StackOverflow, 2018; GitHub, 2017; Kumar and Dahiya, 2017). From these publications, we can see an exponential increase in the importance of scripting languages, for example: JavaScript, Perl, Python, Ruby. This trend especially holds for JS, which has become the easy-to-learn language of the exploding web and IoT. Although there were several experiments to substitute JavaScript with different languages – e.g.: Dart[1], TypeScript[2], CoffeScript[3], Elm[4] – these endeavors did not reach their goal, or at least not completely: they either became used only by a small community or only in a specific area of the web. Thus, JavaScript is still the most popular option for programming the web.

Interpretation is a very typical way of execution for scripting languages. In the domain of JavaScript, interpreters are usually called engines. As JavaScript is core to the contemporary interactive web, everyone who uses social networks, online food ordering services, or internet banking, is using at least one engine built into their browser. However the web is not the only use case of JavaScript, there are machine-to-machine communication scenarios with embedded devices running standalone engines. With the widespread use of JavaScript, its capabilities came into view, both for engine developers and the developers creating JavaScript applications. This is mainly due to the increasing user experience requirements. The capabilities of a JavaScript engine could be improved in several areas: run-time performance, RAM and ROM footprint, features richness, and correctness. However, these capabilities are weighted differently based on the use case: for classic browsing scenarios, performance is the most critical aspect, but for engines in embedded devices footprint is the important factor.

JavaScript engines are able to optimize the application under execution to some extent. The dynamic nature of JavaScript can however make the traditional optimization techniques difficult (Lee et al., 2010). JavaScript developers are able to make the engines

[1]https://www.dartlang.org/
[2]http://www.typescriptlang.org/
[3]http://coffeescript.org/
[4]http://elm-lang.org/

Lóki, G. and Gál, P.
JavaScript Guidelines for JavaScript Programmers - A Comprehensive Guide for Performance Critical JS Programs.
DOI: 10.5220/0006918903970404
In *Proceedings of the 13th International Conference on Software Technologies (ICSOFT 2018)*, pages 397-404
ISBN: 978-989-758-320-9

perform better optimizations and thus the execution speed can be influenced. In statically compiled languages, this can be achieved with different guidelines, however for JavaScript, such guidelines are not common.

Several guidelines for JavaScript have been published addressing this topic (Wilton-Jones, 2006; Zakas, 2009a; Zakas, 2009b; Herczeg et al., 2009; Herczeg et al., 2012; Osmani, 2012), and most of them overlap each other. One of them presents a concrete comparison of different web browsers on the available JavaScript guidelines at the time (Herczeg et al., 2012) . It reveals several correlations between JavaScript engines and optimal guidelines. In this paper, the legacy guidelines are re-examined and it is measured how they perform in the current version of the JavaScript engines. Additionally, with the evolution of the JavaScript language some of the new constructs are investigated and measured, resulting in new guidelines and optimizations.

The rest of the paper is organized as follows. In Section 2, the guidelines and the reasons for their importance are discussed. The legacy guidelines are also introduced, and some of the new language constructs are presented. In Section 3, the results for the guidelines are discussed and the new guidelines are presented. In Section 4, the related works are presented. Finally, the paper is concluded in Section 5.

2 GUIDELINES

Developers use several kind of guidelines in their development processes: to keep the source code in style, to satisfy naming conventions, to apply different kinds of APIs, and to conform to such software requirements as optimal performance or user experience. Although formating and structure related topics are well known and analyzed by many in general, e.g. (Fard and Mesbah, 2013), performance related guidelines are a neglected area in scripting languages. As we have mentioned, relatively few researches and comparisons focusing on the performance of JavaScript in terms of guidelines have been done.

Before going forward, we are going to define what a guideline is. A guideline is a code transformation which shows how to convert one source code to another while solving a previously defined problem. Guidelines are not binding and are not enforced. In this paper, this means that code transformation is a similar code conversion action on JavaScript code snippets which tries to improve the performance of the container software.

We would like to note that the reason why we are focusing on similar and not identical code transformation is that similar transformation does not limit the usage of code structures and language features as opposed to identical transformation which sometimes impossible or can be very expensive. Of course, in many cases it is possible to create an equivalent code transformation which tries to address the same goal, but in these cases the domain of the transformation will be much narrower. Consequently, we will focus on those similar code transformations which are about to improve the performance.

2.1 Legacy Guidelines

In the previously published researches and blog posts (Wilton-Jones, 2006; Zakas, 2009a; Zakas, 2009b; Herczeg et al., 2009; Herczeg et al., 2012; Osmani, 2012) several guidelines were listed. Most of them were created to speed up execution, but not all of them concentrated on the JavaScript module itself. Many suggestions and guidelines are about to improve the performance of the Web browser instead of the JavaScript engine itself. We are only focusing on the performance topic of JavaScript engines. In the upcoming sub-sections, we present the overview of the legacy guidelines from the previously cited researches.

- *Using Local Variables:* This guideline describes that it is more fruitful to use local variables instead of global ones. The reason behind this guideline is very simple; try to reduce the visibility of a variable. It is a common programming paradigm. The architecture logic behind this is that a complex lookup method is called each time the global variable is accessed, and this lookup traverses the whole scope chain every time.
- *Using Global Static Data:* There is an exception to the previously defined *Using Local Variables* guideline. This exception is when the developer wants to introduce a large global static object. In this case, if one is about to use the large static data as a local variable it will be constructed every time when the data is accessed. This guideline suggests avoiding to create the same large object in every usage even if it is a local data. It is better to use a global one instead.
- *Caching Object Members:* It is a general use-case in the development process to access a complex object structure within a loop. In this case, it is possible to save the object field lookup if it is stored in a local variable. The reason for this is similar to the explanation given at *Using Local Variables* guideline.

- *Avoiding With:* The JavaScript language contains a `with` language construct which adds an extra context on the top of the lookup scope chain. The engines first try to lookup for the requested variables within the extra scope contexts, and after that doing the regular lookup. So, if the requested variables are not in the extra scope chains, the lookups will do extra work anyway. This can be avoided if the developers do not use the `with` language construct.

- *Creating Objects:* This guideline suggests avoiding to create objects like the developers do in object-oriented languages. The reason behind this is that creating an object in an object-oriented way invokes a new function creation with a function call as well. This could easily degrade the performance, especially when doing this within a loop.

- *Avoiding Eval*: As the JavaScript developers say: “all the evil and all the possibility comes from `eval`” (Kovalyov, 2015). The `eval` construct in JavaScript is a function which parses the string parameter of the function, and executes the content as a JavaScript at the place of the execution. Its purpose is to evaluate the additional dynamical JavaScript code on-the-fly, but every time is it is used it is a potential opening for someone to inject harmful code into the JavaScript application. However, there are a some legitimate use cases for the `eval` function, but they have their price in terms or performance and memory consumption.

- *Function Inlining:* A very general optimization for non-script languages (Muchnick, 1997), but not for JavaScript. Since none of the engines does control flow related compiler optimization. Function inlining optimization is a missing algorithm in the execution engines, but it can be done by hand. With this guideline, one can save the function calls instructions for the inlined functions.

- *Common Sub-expression Elimination:* This is a well known compiler optimization (Muchnick, 1997) which can be done in JavaScript as well. The logic is to store the result of a common expression in a local variable and reuse the created local variable within all of the representing common expression places.

- *Loop Unrolling:*This guideline is also a general compiler optimization (Ueberhuber, 1997). Loop unrolling is a loop transformation technique that helps to optimize the execution time of the application by removing or reducing iterations. This technique increases performance by eliminating loop control and loop test instructions.

2.2 ECMAScript 6-based Guidelines

The ECMAScript 6 standard (Ecma International, 2015) was introduced in 2015. Since then, the main web browsers and JavaScript engines have been adopting its features. This is true for JavaScript engines targeting the embedded domain as well. Nowadays, we can see lots of support of ECMAScript 6 in the world of JavaScript engines. The main purpose of introducing ECMAScript 6 was to improve utility, and move JS closer to the actual desktop languages, and facilitate the use of the language for every web developer. On the other hand, the characteristics of the ECMAScript 6 have not been examined in details. One of the most important topics is totally missing: there is no such analysis which compares ECMAScript 6 and 5 features in terms of performance.

To validate and introduce possible new guidelines we have analyzed and evaluated the following ECMAScript 6 features and constructs. Our goal is not only to introduce new guidelines which can help developers improve the performance of their application, but to show if a new language construct changes performance compared to the ECMAScript 5 version one.

- *Arrow Function*: The arrow function is an expression which has a shorter syntax than the function expression and does not have its own this, arguments, super constructs. Many developers’ discussions suggest using arrow function if a non-method function is needed (Figures 1 and 2).

```
array.map(function(it){
  return it * local_var;})
```

Figure 1: ECMAScript 5 Arrow Simulation.

```
array.map(it => it * local_var)
```

Figure 2: ECMAScript 6 Arrow.

- *Class Definition:* The `class` is a ”special function” in JavaScript. It has the same syntax as anyone can define for function expression and declaration. The main motivation was to move the JavaScript language closer to the object-oriented programing languages (Figures 3 and 4).

- *Enhanced Object Properties*: Object literals are extended to support setting the prototype for constructions, shorthands for assignments, defining methods, making super calls, and computing property names with expressions. This brings object literals closer to class definition (Figures 5 and 6).

```
var cat = function(name) {
  this.name = name;
  this.speak = function () { v++ }
}
var lion = function(name) {
  parent = new cat(name);
  this.speak = function() {
    parent.speak(); v++;
} }
```

Figure 3: ECMAScript 5 Class Simulation.

```
class Cat {
  constructor(name) {
    this.name = name;
  }
  speak() { v++; }
}
class Lion extends Cat {
  speak() { super.speak(); v++; }
}
```

Figure 4: ECMAScript 6 Class.

```
var car = {
  make: make, value: value,
  dep: function dep() {
    this.value -= 2500;
} }
car['make' + make] = true;
```

Figure 5: ECMAScript 5 Enhanced Object Properties Simulation.

```
var car = {
  make, value,
  ['make' + make]: true,
  dep() { this.value -= 2500; }
}
```

Figure 6: ECMAScript 6 Enhanced Object Properties.

- *Template Strings*: Template strings provide an easy to use syntax to create different strings from a previously defined template. Many languages use similar kinds of template strings such as Linux's Bash, C#, Perl or Python. The motivation behind this feature was to extend JavaScript with a well-accepted template constructions from other languages (Figures 7 and 8).

```
'abc '+(isOK()?'':(test?'def':'123'))
```

Figure 7: ECMAScript 5 Template Strings Simulation.

```
`abc ${isOK()?'':(test?'def':'123')}`
```

Figure 8: ECMAScript 6 Template Strings.

- *Tagged Templates*: A more advanced form of template literals are tagged templates. Tags allow to parse template literals with the help of a function. The helper function returns the manipulated string using the input as a string array and the variables as additional value parameters (Figures 9 and 10).

```
myTag({0:"that ",1:" is a "},person,age)
```

Figure 9: ECMAScript 5 Tagged Templates Simulation.

```
myTag`that ${person} is a ${age}`
```

Figure 10: ECMAScript 6 Tagged Templates.

- *Destructing Objects:* In the JavaScript world destructing objects is a fail-soft action to unbind values from its container. In ECMAScript 6 features, this is about to unpack values from array, or properties from objects, into district variables. This could be very handy for developers in many programing situations. Similar language features can be seen in other scripting languages (such as Python) (Figures 11 and 12).

```
a=10; b=20;
var _ref=[10,20,30,40,50];
a=_ref[0]; b=_ref[1];
rest=_ref.slice(2);
```

Figure 11: ECMAScript 5 Destructing Objects Simulation.

```
[a,b]=[10,20];
[a,b,...rest]=[10,20,30,40,50];
```

Figure 12: ECMAScript 6 Destructing Objects.

- *Spread Operator*: In ECMAScript 6, an extended parameter handling has been introduced. The most important one is the spread operator which spreads the elements of an iterable collection (like an array or even a string) into both literal elements and individual function parameters (Figures 13 and 14).

```
function f(x, y, z) {
  return x + y + z;
}
var a = [1, 2, 3];
f(a[0],a[1],a[2]);
```

Figure 13: ECMAScript 5 Spread Operator Simulation.

```
f(...a);
```

Figure 14: ECMAScript 6 Spread Operator.

- *Constants*: One of the most noticeable change in ECMAScript 6 is that it is possible to create constant values in JavaScript. The `const` construct is defined to hold only constant values. In ECMAScript 5, only the values stored in the global scope can be configured to be constant.
- *Iterators*: This feature allows objects to customize their iteration behavior. Additionally, it supports "iterator" protocol to produce a sequence of values, and provide a convenient method to iterate over all values of an iterable object (Figures 15 and 16).

```
for(var i=0,n=array.length;i<n;i++)
  array[i]
```

Figure 15: ECMAScript 5 Iterators Simulation.

```
for (var value of array)
  value
```

Figure 16: ECMAScript 6 Iterators.

- *Generators*: Many languages contain generators and the `yield` construct. The same functionality has been added to the ECMAScript 6 feature set. Generators are the subtypes of iterators which include additional `next` and `throw` functions. These enable values to flow back into the generator, so the `yield` can return with the next value (Figures 17 and 18).

```
function foo(param){
  this.param = param;
  this.next = function() {
    var res;
    var done = true;
    if (param >= 1) {
      res = param;
      param = param - 1;
      done = false;
    }
    return {value:res, done:done};
} }
```

Figure 17: ECMAScript 5 Generators Simulation.

```
function* foo(param){
  while (param >= 1) {
    yield param;
    param = param - 1;
} }
```

Figure 18: ECMAScript 6 Generators.

- *Map Structure*: In ECMAScript 6, several efficient data structures for common algorithms have been introduced (for example: Map, Set, WeakMap, WeakSet). Since these were not part of the ECMAScript 5 standard, function objects were used to hold the same functionality previously. In our evaluation, we focus on Map structures, since the main logic is similar to the other as well, e.g. Map (Figures 19 and 20).

```
m["abc"] = 123
m[576]
"abc" in m
```

Figure 19: ECMAScript 5 Map Structure Simulation.

```
m.set("abc", 123)
m.get(567)
m.has("abc")
```

Figure 20: ECMAScript 6 Map Structure

- *Symbols:* In ECMAScript 6, there is a new feature called `Symbol` which is global symbol, indexed through unique keys. Every symbol value returned from `Symbol()` is unique. As the simulated ECMAScript 5 implementation is long, the paper omits presenting the examples for this feature.
- *Binary Literals:* Now, in ECMAScript 6, it is possible to enter binary literals. ECMAScript 5 only provided numeric literals in octal, decimal, and hexadecimal form. The new standard added support to describe numbers in binary and another octal form as well (Figures 21 and 22). This can help developers when they are representing numbers for binary operations (such as binary `or`, `xor`, `and`, and negation).

```
parseInt("111110111",2)===503
parseInt("767",8)===503
```

Figure 21: ECMAScript 5 Binary Literals Simulation.

```
0b111110111 === 503
0o767 === 503
```

Figure 22: ECMAScript 6 Binary Literals.

3 RESULTS

In this section, the different guidelines are compared. The measurement was done on a Raspberry Pi 3 Model B. The reason why we choose this hardware as the only measurement platform is that we wanted to choose a platform which all of the JavaScript engines can be applied to, and it is placed between the two main sectors, the embedded and desktop world. The advantage of desktop engines is the performance at the expense of code size and memory consumption. The restricted environment cannot pay this price. So, the restriction of the above mentioned hardware fits into the embedded world, and it still allows to execute the desktop engines without kill all performance optimizations.

We used the following hardware and software environment: BCM2835 ARMv7 Quad Core CPU, 1GB DDR2 memory, 4GB Class 10 SD card on a Raspbian GNU/Linux 8.0 (jessie) OS with a Linux raspberrypi 4.9.35-v7+ kernel image. The measurement framework was written using Python 2.7.9 and Bash scripts. The measurement methodology was the following:

- Each legacy guideline has an original and transformed code snippet.
- Each ECMAScript 6 guideline has an ECMAScript 6 and an ECMAScript 5 version.

- The measurement framework extends every test with utility functions to measure and save the elapsed time within the main of the test cases.
- Since the desktop engines still perform better than the embedded ones, an additional loop iteration was introduced for desktop engines.
- Each measurements has been executed twenty times and the median of the results was used to compute the relative percentages on the figures.

As previously discussed, JavaScript engines are designed for different software and hardware stacks. Several ones work on desktop systems, others focus on the embedded world, and there are some of them which try to focus on both. Due to the different targeting, several features might be missing or not implemented in one or an other JavaScript engine, or even working on a totally different way. If one is interested in which functionalities are supported by JavaScript engines there are different online comparison tables showing the implemented and supported features [5].

On the other hand in the JavaScript engines which are targeting at the low-end hardwares or focusing on supporting machine to machine communication it is not so evident to support all JavaScript language feature. Our evaluation shows that these embeddable engines do not support the full spectrum of ECMAScript 6 language constructs. There is no single language construct which is supported by all the three embeddable engines (see below), so we cannot do a conclusive evaluation with these engines.

In this paper, six JavaScript engines are examined:

- *JavaScriptCore*: it is the JavaScript engine of Safari and WebKit-based web browsers. You can find it within iPhone, and Mac desktop machines. (Version: ToT 2017-12-10)
- *V8*: it is the main JavaScript engine of Google's Chrome and Chromium-based web browsers as well. Most smart phones delivered with Android OS have it, and of course, the common desktop machines can use Google's Chrome web browser. (Version: 6.4.99)
- *Spidermonkey*: this engine is used by Mozilla's Firefox web browser. Firefox can run within phones, tables, and desktop machines. (Version: 59)
- *JerryScript*: this engine is the first in this row which is targeted only at the embedded world. This engine is developed by Samsung, Intel, ARM, and the University of Szeged with other open source community members. You can see it running inside several smart watches. (Version: 1.0 - da24727)
- *Espurino*: it is a JavaScript interpreter for micro-controllers. The supporting company also built up a hardware stack around the software. (Version: 1v95)
- *Duktape*: A small footprint, easily embeddable, ECMAScript E5/E5.1 engine. (Version: 2.2.0)

In this paper, our target is to evaluate JavaScript guidelines to see how they are affecting the different engines. For each language construct and feature the run-time was measured with, and without guidelines and a resulting percentage was calculated in a way that the run-time of the faster guideline was divided by the run-time of the slower one.

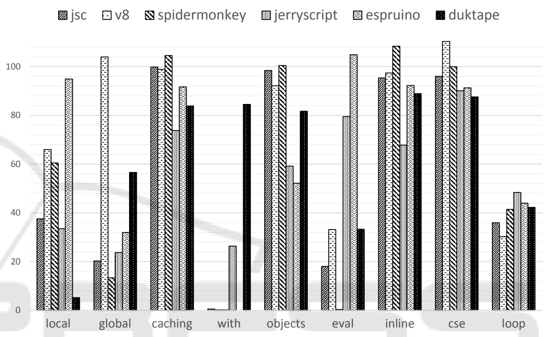

Figure 23: Performance Improvement with Legacy Guidelines.

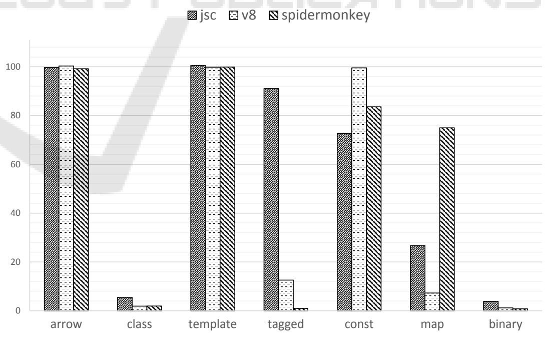

Figure 24: Performance Improvement with ECMAScript 6.

The *Figure 23* shows that the legacy guidelines are still valid, and it is worth using them in terms of performance.

The evaluation of ECMAScript 6 shows some unexpected and some very significant changes from ECMAScript 5. We have defined two groups; one where the old standard performs better (*Figure 25*) and the other one where the new standard has faster code paths in the engines (*Figure 24*). Based on the results, we can define our guidelines for ECMAScript 6 language features as such:

[5] https://kangax.github.io/compat-table/es6/

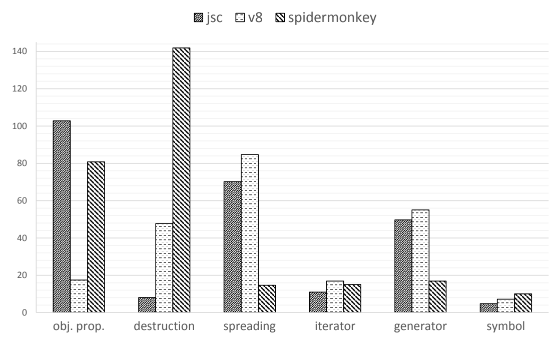

Figure 25: Performance Improvement with ECMAScript 5.

- *Arrow Function:* Use the arrow functions if a non-method function is needed. The reason for this that although JavaScript engines reveal very small performance improvement, the ECMAScript 6 form is clearer and easier to adopt.
- *Class Definition:* Use the `class` definition. The reason for this is that most of JavaScript engines perform better by using fast path implementations. The new standard can save even 95% run-time.
- *Enhanced Object Properties:* Do not use the enhanced object properties in the ECMAScript 6 form. Use the ECMAScript 5 variant instead. Results show that significant speed-up can be seen with most execution engines if the old standard is used.
- *Template Strings:* There are no significant changes when using the ECMAScript 5 or 6 version of the template strings, so there is no clear conclusion about this construct. In this case, we suggest following the new standard. The engines may improve this code later.
- *Tagged Templates:* Use tagged templates. The engines implement a special code path for this construct. One of the engines over-performs the others with a more than 99% run-time improvement.
- *Destructing Objects:* Do not use the destructing construct. The reason for this is that it significantly slows down almost all engines - except one.
- *Spread Operator:* Do not use the spread operator. Most of the JavaScript engines perform better when using the ECMAScript 5 form.
- *Constants:* Use constants construct. All engines perform better with `const`. A special code path have been implemented for this.
- *Iterators:* Do not use iterators. The reason for this is that the ECMAScript 5 form is still faster. Currently, there is no fast path implementation for this construct in the engines.
- *Generators:* Do not use generators. The reason for this is the same as in the iterators case.
- *Map Structure:* Use the new built-in structure, e.g. Map construct. The engine implemented these features with a fast code path.
- *Symbols:* Do not use the new symbols standard. Although JavaScript engines have a new code path for this feature, simulating it is currently faster.
- *Binary Literals:* Use the binary literals. The reason for this is that the new feature implementations are very fast, and there is no need to call any parsers to read binary literals.

4 RELATED WORKS

Although the topic of writing efficient JavaScript applications and code snippets is very important for the software industry, the main area is to evolve the JavaScript software stack is the improvement of the engines themselves. Based on the well-studied research area in static languages (Nielson et al., 1999; Torczon and Cooper, 2011) the static optimization algorithms can be the first good choice to use them in the JavaScript engines as well. However the JavaScript is a dynamic language where the static optimization algorithms cannot determine various properties, for example the types of the objects, variables, or even the structure of the input script. For this challenges, user intervention is needed, for example applying guidelines to improve performance.

There is only a limited number of studies which discuss how to improve one or the other characteristics of a JavaScript application with source transformations. There were studies on how to transform JavaScript projects to look like an object-oriented source code (Silva et al., 2015; Silva et al., 2016), but the focus of these researches was to improve the maintainability, and not to improve the user experiences (such as performance, or memory consumption). Another approach could be to analyze what the best practices are for JavaScript (Ölund and Karlsson, 2016). For ECMAScript 5 features there are only a few related papers measuring the run-time effect of the guidelines (Herczeg et al., 2009; Herczeg et al., 2012).

5 SUMMARY

In this paper, we have evaluated available guidelines for JavaScript engines, and presented new ones targeting at the ECMAScript 6 feature set. The presented results show that the guidelines are still important, and one can get significant performance improvement adapting them in a JavaScript project. Although the results are very conclusive now, it is very advisable to revisit and evaluate the importance of the guidelines from time to time. As the JavaScript engines evolve, it might happen that some of the guidelines become obsolete.

A follow-up work can be to evaluate the importance of the guidelines in terms of memory consumption, generated code size and even in energy consumption in the embedded world. We are lack of these kind of information which can help the JavaScript projects independently and engine developers to improve the characteristics of the execution engines as well.

ACKNOWLEDGEMENTS

This research was supported by the Hungarian Government and the European Regional Development Fund under the grant number GINOP-2.3.2-15-2016-00037 (“Internet of Living Things”).

REFERENCES

Ecma International (2015). *ECMAScript 2015 Language Specification*. Geneva, 6th edition.

Fard, A. M. and Mesbah, A. (2013). JSNOSE: Detecting JavaScript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

GitHub (2017). Open source survey. http://opensourcesurvey.org.

Herczeg, Z., Loki, G., Szirbucz, T., and Kiss, A. (2009). Guidelines for JavaScript programs: Are they still necessary? In *Nordic Workshop on Model Driven Software Engineering*, pages 59–71.

Herczeg, Z., Loki, G., Szirbucz, T., and Kiss, A. (2012). Validating JavaScript guidelines across multiple web browsers. *Nordic Journal of Computing*, 15:8–31.

Kovalyov, A. (2015). *Beautiful JavaScript*, chapter 2. OReilly Media.

Kumar, K. and Dahiya, S. (2017). Programming languages: A survey. *International Journal on Recent and Innovation Trends in Computing and Communication*, 5(5):307–313.

Lee, S.-W., Moon, S.-M., Jung, W.-K., Oh, J.-S., and Oh, H.-S. (2010). Code size and performance optimization for mobile JavaScript just-in-time compiler. In *Proceedings of the 2010 Workshop on Interaction between Compilers and Computer Architecture*.

Muchnick, S. S. (1997). *Advanced compiler design implementation*. Morgan Kaufmann Publishers.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). Principles of program analysis.

Ölund, H. and Karlsson, J. (2016). Investigation of the key features in ECMAScript 2015.

Osmani, A. (2012). How to write fast, memory-efficient JavaScript. https://www.smashingmagazine.com/2012/11/writing-fast-memory-efficient-javascript/.

Silva, L. H., Hovadick, D., Valente, M. T., Bergel, A., Anquetil, N., and Etien, A. (2016). JSClassFinder: A tool to detect class-like structures in JavaScript. *CoRR*, abs/1602.05891.

Silva, L. H., Ramos, M., Valente, M. T., Bergel, A., and Anquetil, N. (2015). Does JavaScript software embrace classes? In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 73–82.

StackOverflow (2018). Stack Overflow annual developer survey. https://insights.stackoverflow.com/survey/2018.

Torczon, L. and Cooper, K. (2011). *Engineering A Compiler*. Morgan Kaufmann Publishers, 2nd edition.

Ueberhuber, C. W. (1997). *Numerical computation: methods, software, and analysis*. Springer.

Wilton-Jones, M. (2006). Efficient JavaScript. https://dev.opera.com/articles/efficient-javascript/.

Zakas, N. C. (2009a). Speed up your JavaScript: The talk. https://www.nczonline.net/blog/2009/06/05/speed-up-your-javascript-the-talk/.

Zakas, N. C. (2009b). *Writing Efficient JavaScript – Even Faster Websites*, chapter 7. OReilly Media.