Guidelines for JavaScript Programs: Are They Still Necessary?

Zoltán Herczeg, Gábor Lóki, Tamás Szirbucz, and Ákos Kiss

Department of Software Engineering, University of Szeged
Honvéd tér 6., H-6720 Szeged, Hungary
{zherczeg,loki,akiss}@inf.u-szeged.hu, szirbucz.tamas@stud.u-szeged.hu

Abstract. Nowadays, JavaScript is the language for developing dynamic websites. Before the age of JIT engines, several guidelines were published about how to write efficient JavaScript code. Our research focuses on whether programmers should still adhere to these guidelines or can they rely on the JIT compiler to achieve good performance results. In this paper, we present the experiments where we investigate the interaction of just-in-time compilation and programming guidelines on JavaScript execution performance. We present sometimes surprising observations, and conclude that the importance of guidelines does not decrease with the introduction of JIT technology.

1 Introduction

Although, in the past, there were several choices for client-side scripting of web pages, like JavaScript, VBScript, PerlScript, and even Tcl/Tk [1], there is no doubt that nowadays JavaScript is the language for developing dynamic websites. This happened that way even though JavaScript is not the ideal programming language. Being an interpreted language, it is usually considered slow for complex tasks, and the cross-browser problems do not make the life of JavaScript programmers easier either. However, new solutions are emerging for these shortcomings. In 2006, Google Web Toolkit (GWT) [2] was released to ease the development of browser-independent JavaScript applications. In addition, the JavaScript engines of the browsers became faster as well, mostly by introducing just-in-time (JIT) compilation techniques [3]: Mozilla came with TraceMonkey [4], WebKit introduced its SquirrelFish Extreme [5], and Google announced Chrome with its V8 JavaScript engine [6].

However, before the age of JIT engines, several guidelines were published about how to write an efficient JavaScript code [7–9]. Our research focuses on whether programmers should still adhere to these guidelines or can they rely on the JIT compiler to achieve good performance results – as programmers do rely on classic compilers to generate optimal code in the case of static languages, like C. In this paper, we present the experiments where we investigate the effect of just-in-time compilation and programming guidelines on JavaScript execution performance.
The structure of the rest of the paper is as follows: In Section 2, we present the most well-known coding guidelines for JavaScript, and give a short explanation for them. In Section 3, we discuss how these guidelines could – or could not – be automatized (i.e., turned into optimizations), what the gain would be, and what the barriers are. In Section 4, we present the key result of our paper, the execution speedups achieved by adhering to programming guidelines, even if just-in-time compilation is applied. In Section 5, we give an overview on related work, while in Section 6, we present our conclusions.

2 Optimization Guidelines

There are several guidelines to aid the JavaScript programmers in writing fast (or faster) code. Some of these guidelines are based on well-known static compiler optimization techniques, while others focus on JavaScript language specialities. In the following subsections, we give an overview of these techniques.

2.1 Using Local Variables

Every time a variable is accessed in JavaScript, a complex lookup method is called that involves searching through the whole scope chain. However, all execution engines are known to speed up the lookup of local variables. Thus, guidelines suggest using local variables instead of global ones whenever possible (see Fig. 1).

\[
\text{for (i = 0; i < 10000000; ++i) ;}
\]

(a)

\[
\text{for (var i = 0; i < 10000000; ++i) ;}
\]

(b)

Fig. 1. Using local variables instead of global ones.

2.2 Using Global Static Data

In general, guidelines suggest using local variables instead of global ones, as explained in the above section. There is one exception to this rule, however. In most static languages, it is possible to define variables inside functions whose lifetime spans across the entire run of the program, called static variables, which are often constants, too. However, JavaScript does not support the concept of either constants or static variables, and initializations are nothing more but assignments.

Thus, if an array literal is used in an assignment, as shown in the example of Fig. 2(a), the array will be constructed every time when the execution reaches the assignment. These superfluous operations can take considerable time – even more than the lookup of a global variable would cost –, thus guidelines suggest using globally initialized variables in this special case, as presented in Fig. 2(b).
function hexDigit(s) {
    var digits = ["0","1","2","3",
                  "4","5","6","7",
                  "8","9","a","b",
                  "c","d","e","f"];
    return digits[s];
}

for (var i = 0; i < 5000000; ++i)
    hexDigit(i & 0xf);

(a) (b)

Fig. 2. Moving static data out of functions.

2.3 Caching Object Members

Whenever the same object members are accessed several times in a script, e.g., in a loop, it is advised to cache the values of the members in local variables, as shown in Fig. 3. The reason for this is similar to the explanation given in Section 2.1; member resolution is expected to be slower than local variable lookup.

var o = {a: 678,b: 956}
var r
for(var i=0;i<3000000;++i)
    r = o.a + o.b

var o = {a: 678,b: 956}
var r
var ca = o.a
var cb = o.b
for(var i=0;i<3000000;++i)
    r = ca + cb

(a) (b)

Fig. 3. Caching object members in variables.

2.4 Avoiding With

The with language construct of JavaScript adds a computed object to the top of the scope chain and executes its body with this augmented scope chain. It is a very helpful feature if the chain of object references or the name of the object is very long, but in practice it increases the execution time. Again, guidelines suggest that better performance result can be achieved if local variables are used for accessing object members instead of with statements (see Fig. 4).

2.5 Creating Objects

The most important suggestion of guidelines about object creation is to avoid creating objects like in object-oriented (OO) languages, since this kind of object creation has to be solved through a function call. It is proposed to use the JavaScript Object Notation (JSON) form, which allows specifying object literals in the script code. In Fig. 5, some possible object creation approaches are presented: in subfigure (a), the object creation is implemented in OO way, while subfigure (b) shows an inlined object creation solution, and subfigure (c) gives a JSON-based object creation example.
2.6 Avoiding Eval

The eval function evaluates a string and executes it as if it were script code. This language feature can help hiding or obfuscating the script code, and can also help executing dynamic script code, but it has its own cost. Each string that is passed to the eval function has to be parsed and executed on-the-fly. This cost has to be paid every time the execution reaches an eval function call. So, trying to avoid eval is considered as a good idea whenever there is an alternative solution, as shown in Fig. 6.

2.7 Function Inlining

Function inlining is a traditional compiler optimization technique [10] that replaces a function call with the body of the called function. In JavaScript, performing a function call is an expensive operation. It takes several preparatory steps to perform: allocating space for parameters, copying the parameters, and resolving the function name. With function inlining, as shown in the example in Fig. 7, the cost of these steps can be saved. (For the sake of completeness, in the example we give two function call-based implementations beside the inlined version; subfigure (a) shows the call of a user-defined function, while subfigure (b) utilizes a built-in function.)
function funcS() { return "s" } function funcD() { return "d" } function funcL() { return "l" }

var code = "dsdllsdsdlls";
var len = code.length
var res = ""

for (var j = 0; j < 50000; ++j) {
    for (var i = 0; i < len; ++i)
        res += eval("func" + code.charAt(i) + "()")
}

for (var i = 0; i < 2000000; ++i)
    get_roots(i & 0xff, i & 0x7, 10)

var a = abs(4000000-i);
var a = Math.abs(4000000-i);
var a = (4000000-i) >= 0 ? (4000000-i) : -(4000000-i);

2.8 Common Sub-expression Elimination

Common sub-expression elimination (CSE) is another performance-targeted compiler optimization technique [10] that searches for instances of identical expressions and replaces them with a single variable holding the computed value. In the guidelines, this is suggested to be done manually (see Fig. 8), since a typical JavaScript engine does not support this optimization. Using a single local variable for a common sub-expression is expected to be always faster than leaving the code unchanged.

Fig. 6. Avoiding eval.

Fig. 7. Function inlining

Fig. 8. Common sub-expression elimination.
2.9 Loop Unrolling

Loop unrolling [11] is yet another compiler optimization technique (Fig. 9) that is suggested by the guidelines to be applied manually. It is most effective if the loop body is small but the loop runs long. The performance gain comes from the absence of most of the loop test and increment instructions.

```javascript
var iterations = 100000000
var counter=0
for(i=iterations;i>0;--i) {
    counter++
}
```

```javascript
var iterations = 100000000
var counter=0
var n = iterations % 8
if (n>0)
    do {
        counter++
    }
    while (--n)
    n = iterations >> 3
if (n > 0)
    do {
        counter++
        counter++
        counter++
        counter++
        counter++
        counter++
        counter++
    }
    while (--n)
```

(a) (b)

Fig. 9. Loop unrolling.

2.10 Optimizing Loop Indices

Regarding loop indices, guidelines typically make two recommendations. First, post-increment and decrement operators on the loop index variable should be replaced by pre-increment and decrement operators, if possible. Second, decrementing a loop should be preferred over incrementing it.

The rationale behind the first recommendation is that post-operators take more machine instructions to execute than pre-operators, but JavaScript engines usually do not transform post-increment and decrement operators automatically to their ‘pre’ counterpart whenever possible, i.e., when the value returned by the operation is unused. Secondly, decrementing a loop is preferred over incrementing it since comparison to zero is much faster on almost every architecture than comparison to any other number.

2.11 HTML DOM

Almost every guideline contains suggestions for optimizing HTML Document Object Model (DOM) based object accesses, e.g., dynamic HTML generation. The most typical recommendation is not to access DOM objects too frequently, since the DOM bindings are expected to be slow. However, these guidelines are less JavaScript language feature-related and more browser-specific, thus they are not in the scope of this paper. Therefore, we do not discuss them further.
3 Static Optimization Difficulties

The guidelines in the previous sections give directions for JavaScript programmers on how to write effective code. It would be very convenient however, if these performance speed-up techniques would not need to be applied manually, but could be turned to automatic code transformations, i.e., compiler optimizations. The domain of compiler optimizations is a well-studied research area [10, 12–15]. The experience with static languages is that optimization algorithms are worth to apply, since the price of the techniques is only to be paid at compilation time, and the gain in performance is considerable. Thus, the need for optimization algorithms naturally rises for dynamic languages as well, and the already-existing guidelines could act as natural starting points for designing these techniques. However, as we will see below, the language features of JavaScript make all static optimization techniques ineffective.

First, let’s consider Fig. 10. The loop in function test1 is supposedly infinite, which continuously prints “Hello World!” messages. However, the loop in the example stops after three iterations, because of the parameter used in the given function call. That parameter is passed to eval and there it redefines the print identifier from the built-in function to a user-defined one. Moreover, since the new implementation of print is defined inside the scope of the test1 function, it can access its local variables as well. (And, since the loop index variable is incremented every time print is invoked, the loop will terminate in this case.)

```javascript
function test1(cmd)
{
    var a = 0
    eval(cmd)
    while (a < 3)
        print("Hello World!")
}

test1("var pr = print; print = function(text) { a++ ; pr(text) } ")
```

Fig. 10. Eval, function redefinition, and access to local variables.

The example in Fig. 11 produces the same output as Fig. 10, but achieves it in different ways. The code shows that one does not have to use eval to get hardly predictable results. In the example, a setter function is used to turn assignments to variable b into function calls. Similarly to the previous example, the local variables of function test2 can be accessed in the called function, too. Additionally, as the example shows, the definition of the setter method can be obfuscated; it is done via the def function call in this case. Thus, theoretically, any function call can be a setter function.

The last example in Fig. 12 shows an uncommon usage of valueOf. The valueOf method of an object is implicitly called when an operator requires the primitive value of an identifier. Thus, in this case the loop test implicitly increases the loop index. Unfortunately, this effect is completely invisible to a static analyzer of the test3 function.
var def = __defineSetter__
function test2(name) {
    def(name, function(value) { print("Hello world!"); a++ })
    for (a = 0; a < 3; /* Do nothing */) {
        var a
        b = void(0)
    }
}
test2("b")

Fig. 11. Setter function.

var x = 0
Number.prototype.valueOf = function() { return x++ }
function test3(a) {
    while (a < 3)
        print("Hello world!")
}
test3(new Number(0))

Fig. 12. Overriding the valueOf() method.

The above examples show several unpredictable changes that can happen to variables and functions, that static optimization algorithms cannot foresee. Since compiler optimizations always have to be safe, these language features make the application of complex optimization algorithms, automatized programming guidelines to JavaScript practically infeasible. Thus, it seems that guidelines remain only guidelines. In the following section, we investigate how effective they are in the case of state-of-the-art JavaScript engines.

4 Measurements

In this section, we present the results of applying the programming guidelines presented in Section 2. All data was measured using the JavaScript execution engine of WebKit revision 43818 on a Dual Core AMD Opteron(tm) Processor 275 with 4 GB memory. We performed two kinds of measurements based on different JavaScript execution models. First, we executed all the examples of Section 2 in interpreter mode – this execution model is available on all architectures, and thus, most JavaScript engines support it. Next, we let all examples run in the JavaScript engine using JIT compilation technique.

Although this paper is not focused directly on the performance of the different execution models, we have to note that the JIT execution model in WebKit is 1.5–3 times faster than the interpreter one. In other dynamic languages similar results are reported using JIT.
4.1 The Effect of the Guidelines

Fig. 13 presents the measurement results in a normalized view. The figure is composed of subcharts where each subchart corresponds to a programming guideline discussed in Section 2. In each barchart, the white bars represent the execution times of the example programs in interpreter mode, while the gray bars correspond to the execution times measured in JIT mode. Each bar is labelled with (a), (b), or (c) – where applicable –, which refer to the example programs in the corresponding subfigures. (In the figures of Section 2, (a) marked the unoptimized code, while (b) and (c) denoted optimized code.) In all subcharts, the interpreted execution time of the unoptimized code was used for normalization.

Looking at these charts, we can make two important observations. First, all the programs that conform to the guidelines are faster than their unoptimized counterparts, irrespectively of the execution model. Thus, this leads to the conclusion that the performance guidelines are still useful, for interpreted and JIT-compiled JavaScript programs alike.

Second, in six cases (for the Local variables, Global static data, Avoiding with, Creating objects, Avoiding eval, Loop unrolling guidelines), the application of the guidelines clearly outperforms JIT technology. That is, simply by adhering to the guidelines and still using an interpreter-based JavaScript engine, we can achieve better execution times than by relying solely on JIT compilation (but not considering the guidelines). For example, the Local Variables (b) example program executed in interpreter mode is more than two times faster than Local Variables (a) in a JIT engine. This can be of high importance for platforms where JIT technology is unavailable.

The observations reveal that although state-of-the-art JIT technology has a very good effect on performance, sometimes a simple guideline can result in a higher speed-up. Thus, programmers should not leave all the optimization work to JavaScript engines.
4.2 The Efficiency of the Guidelines

We also investigated how the efficiency of the guidelines changes when switching execution models. It would be reasonable to think that the significance of the guidelines decreases when JavaScript programs are executed using state-of-the-art JIT technology, but Fig. 14 reveals that this is not the case.

In Fig. 14, we show the execution times of the optimized programs (found in the (b) and (c) subfigures of Section 2) but normalize the interpreter-based and the JIT-based executions separately. (I.e., the interpreted execution time of an optimized program is normalized to the interpreted execution time of its unoptimized counterpart, while the JIT-compiled execution time of the optimized code is normalized to the JIT-compiled execution time of the unoptimized version.) However, we show the results side-by-side. Thus, it is easy to see the significance (and the change in the significance) of the guidelines in both execution models.

It is most interesting that in almost all cases, the effect of the guidelines on performance is higher in JIT mode than in interpreted mode. Thus, it seems that the importance of the guidelines does not decrease but – in contrast with the expectations – it increases!

5 Related Works

Since we are not aware of any previous studies on the effect and efficiency of JavaScript programming guidelines, in this section we present works related to two key topics of the paper: programming guidelines and JIT compilation technology.
5.1 Programming Guidelines

Nowadays, the quality of the source code is getting more and more important [16]. There are many studies and papers about how the quality of the source code influences the appearance tendency of errors and bugs in a software product [17]. This revelation leads many project owners to publish coding guidelines regardless of the language. Hence, guidelines are available for Java [18], PHP [19], Action Script [20] and many other commonly used languages.

The coding guidelines mostly contain forms and transformations of the source code in order to make it more readable and understandable. However, there are several guidelines which suggest code optimization transformations as well. The majority of JavaScript-targeted guidelines belong to the second group [7–9]. These guidelines not only contain code transformations, but usually they provide a way to measure their effect in the user’s browser as well. This way, the users can test the worthiness of the optimizations themselves.

Since performance-targeted coding guidelines are sometimes based on classic compiler optimization techniques, we have to mention them here. Static compiler optimizations have a long history, and several books have been written in the past, which give a good summary of the domain [10, 12–15].

5.2 Just-in-Time Compilation

Currently, there are two distinct approaches for the just-in-time compilation of JavaScript. The relatively new, trace-based approach is used by Mozilla’s TraceMonkey [4]. A trace is a runtime profile of JavaScript code. Machine code is generated only for frequently executed code paths guarded by side exits, which provide fall-back mechanism when the execution flow leaves a code path [21, 22]. Thus, the tracing engine can adapt to the current execution flow.

The classic approach is to generate JIT code for all JavaScript functions. This approach is employed by Google’s V8 [6] and Apple’s SquirrelFish Extreme [5] engines. Although the generated code is not adaptive, the fast paths in the generated code are based on static profiles.

6 Summary

In the title of the paper, we posed the question whether guidelines for JavaScript programs are still necessary? This is a timely question, since in these days, new JavaScript engines are introduced in the web browsers, which apply state-of-the-art just-in-time compilation technology. However, it is important to know whether guidelines compiled for old, interpreter-based engines are still valid for the new solutions. In our paper, we gave an overview of the most important JavaScript coding guidelines and investigated their effect in both interpreter-based and JIT-based environments. Our investigations have led to some, sometimes surprising observations:
Programming guidelines are still useful, both for old and state-of-the-art JavaScript execution engines.

In several cases, the execution of guideline-optimized programs in the interpreter is faster than the execution of their unoptimized versions in the JIT engine.

The efficiency of the guidelines is almost always better in a JIT-based engine than in an interpreter-based one.

Thus, we can give a clear answer to the question: guidelines are still necessary. What is more, they seem to gain importance! Moreover, since static compiler optimizations are not applicable to JavaScript because of language features, as discussed in this paper, we predict that the importance of the guidelines will not decrease in the near future.

Still, there is more work to be done on the field of guidelines. In our current experiments, we investigated only example programs and only one JIT engine. Thus, we plan to repeat the experiments in future research with real-life JavaScript applications and different JIT engines as well.

References

1. Loban, S.: Language choices for client-side scripting (November 2001)
2. Google: Google web toolkit
4. Mozilla Foundation: TraceMonkey (November 2008)
6. Google: Google V8 engine
7. Greenberg, J.: JavaScript optimization