Nordic Journal of Computing

VALIDATING JAVASCRIPT GUIDELINES ACROSS MULTIPLE WEB BROWSERS

ZOLTÁN HERCZEG GÁBOR LÓKI TAMÁS SZIRBUCZ

ÁKOS KISS

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

Abstract. Nowadays, JavaScript is *the* language for developing dynamic websites. Previously, 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 state-of-the-art JavaScript execution engines to achieve good performance results. In this paper, we present the experiments where we validate programming guidelines for JavaScript execution performance across multiple state-of-the-art web browsers. We present our observations, and conclude that the importance of guidelines does not decrease with the introduction of JIT technology.

ACM CCS Categories and Subject Descriptors: D.3.3 [Programming Languages]: Language Constructs and Features — JavaScript

Key words: JavaScript, programming guidelines, just-in-time compilation

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 [Loban 2001], 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) [Google, Inc. 2006] 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 [Krall 1998]: Mozilla came with TraceMonkey [Mozilla Foundation 2008], WebKit introduced its SquirrelFish Extreme [Stachowiak 2008], and Google announced Chrome with its V8 JavaScript engine [Google, Inc. 2008]. Opera Software is developing its Carakan engine [Lindström 2009] and Microsoft is including Chakra in Internet Explorer 9 [Niyogi 2010].

In the past, several guidelines were published about how to write an efficient JavaScript code [Greenberg 2001][Garcia 2005][Zakas 2009]. Our research focuses on whether programmers should still adhere to these guidelines or can they rely on state-of-the-art JavaScript engines to achieve good performance results – as programmers do rely on classic compilers to generate optimal code in the case of static

Received October 3, 2010.

languages, like C. In this paper, as a follow-up of our previous experiments [Herczeg *et al.* 2009], we validate JavaScript guidelines across multiple web browsers.

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 measured effect of programming guidelines across several web browsers. 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 specialties. 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).

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).

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.

```
for (i = 0; i < 10000000; ++i) ;
```

(a)

```
for (var i = 0; i < 10000000; ++i) ;
```

(b)

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

```
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)

```
var digits = ["0","1","2","3",
              "4","5","6","7",
              "8","9","a","b",
              "c","d","e","f"];
function hexDigit (s) {
  return digits[s];
}

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

(b)

Fig. 2: Moving static data out of functions.

```
var o = {a: 678,b: 956}
var r
for(var i=0;i<30000000;++i)
  r = o.a + o.b
```

(a)

```
var o = {a: 678,b: 956}
var r
var ca = o.a
var cb = o.b
for(var i=0;i<30000000;++i)
  r = ca + cb
```

(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).

```
var o = new Object()
o.ext1 = new Object()
o.a = 23
o.ext1.ext2 = new Object()
o.ext1.b = 19
o.ext1.ext2.c = 36

with (o) {
  with (ext1) {
    with (ext2) {
      for(var i=0;i<2000000;++i)
        a = b + c
    }
  }
}
```

(a)

```
var o = new Object()
o.ext1 = new Object()
o.a = 23
o.ext1.ext2 = new Object()
o.ext1.b = 19
o.ext1.ext2.c = 36

var ext1 = o.ext1
var ext2 = ext1.ext2
for(var i=0;i<2000000;++i)
  o.a = ext1.b + ext2.c
```

(b)

Fig. 4: Avoiding *with* statements.

```
function create() {
  this.str = "String"
  this.int = 56
  this.num = 6.7
  this.get = function()
    { return this.int }
}
var object
for(var i=0;i<3000000;++i)
  object = new create()
```

(a)

```
var object
for(var i=0;i<3000000;++i) {
  object = new Object()
  object.str = "String"
  object.int = 56
  object.num = 6.7
  object.get = function()
    { return this.int }
}
```

(b)

```
var object
for(var i=0;i<3000000;++i) {
  object = {
    str: "String",
    int: 56,
    num: 6.7,
    get: function()
      { return this.int }
  }
}
```

(c)

Fig. 5: Creating objects.

```
function funcs() { return " " }
function funcd() { return "." }
function funcl() { return "_" }

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)+"()")
}
```

(a)

```
function funcs() { return " " }
function funcd() { return "." }
function funcl() { return "_" }

var code = "dsdllsdsdlls";
var len = code.length
var res = ""
for (var j = 0; j < 50000; ++j) {
  for (var i = 0; i < len; ++i)
    switch (code.charAt(i)) {
    case 's' : res += funcs() ; break
    case 'd' : res += funcd() ; break
    case 'l' : res += funcl() ; break
    }
}
```

(b)

Fig. 6: Avoiding *eval*.

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.

```
function abs(a) {
  return a>=0 ? a : -a
}
var a
for(var i=0;i<8000000;++i)
  a = abs(4000000-i);
```

(a)

```
var a
for(var i=0;i<8000000;++i)
  a = Math.abs(4000000-i);
```

(b)

```
var a
for(var i=0;i<8000000;++i)
  a = (4000000-i) >= 0 ?
        (4000000-i) :
       -(4000000-i);
```

(c)

Fig. 7: Function inlining

```
function get_roots(a, b, c) {
  var ret = {
    x1:((-b+Math.sqrt(b*b-4*a*c))
        /(2*a)),
    x2:((-b-Math.sqrt(b*b-4*a*c))
        /(2*a))
  }
  return ret
}

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

(a)

```
function get_roots(a, b, c) {
  var sq = Math.sqrt(b*b-4*a*c);
  var ret = {
    x1: ((-b + sq) / (2*a)),
    x2: ((-b - sq) / (2*a))
  }
  return ret
}

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

(b)

Fig. 8: Common sub-expression elimination.

2.7 Function Inlining

Function inlining is a traditional compiler optimization technique [Muchnick 1997] 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.)

2.8 Common Sub-expression Elimination

Common sub-expression elimination (CSE) is another performance-targeted compiler optimization technique [Muchnick 1997] 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.

2.9 Loop Unrolling

Loop unrolling [Ueberhuber 1997] is yet another compiler optimization technique (Fig. 9) that is suggested by the guidelines to be applied manually. It is most

```
var iterations = 100000000
var counter=0

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

(a)

```
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++
    counter++
  } while (--n)
```

(b)

Fig. 9: Loop unrolling.

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.

2.10 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, 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 [Muchnick 1997][Nielson *et al.* 1999][Morgan 1998][Cooper and Torczon 2004][Allen and Kennedy 2002]. 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*

```
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.

```
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.

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.)

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 use of *valueOf*. The *valueOf* method of an object is implicitly called when an operator requires the primitive

```
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.

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.

The above examples show several 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 have to remain guidelines only. 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 effect of the programming guidelines presented in Section 2 on JavaScript execution performance. All data was measured on an Intel Pentium T2330 dual-core processor running at 1.6 GHz and equipped with 1 GB of memory. For the experiments, we used the latest available versions of the web browsers most common at the time of writing this paper. Namely, we used (with the code name of the JavaScript execution engine given in parentheses, where known): Apple Safari r50495 nightly build (SquirrelFish Extreme/Nitro), Google Chrome 3.0.195.27 (V8), Mozilla Firefox 3.6 Beta 1 (TraceMonkey), Opera 10.10 Beta (Futhark), and Microsoft Internet Explorer 8.0.6001.18702IC.

Fig. 13 and Fig. 14 present the measurements in a normalized view. The figures are composed of subcharts where each subchart corresponds to the effect of a programming guideline on a web browser. Each row denotes one of the web browsers mentioned above and each column corresponds to a guideline discussed in Section 2.

In each subchart, the bars represent the execution time of the example programs from Section 2. Each bar is labeled 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 original unoptimized code, while (b) and (c) denoted code with guidelines applied.) In all subcharts, the execution time of the original unoptimized code (a) was used for normalization.

The inspection of the results reveals that the efficiency of the guidelines varies heavily across browsers. Moreover, only two browsers, Firefox and Internet Explorer profit from all of the guidelines. In the other three browsers, the use of built-in functions causes a clear performance loss (for Safari and Chrome, the loss was so high that we were not able to plot the bars correctly) and the different object creation techniques bring no definite gain either. (In the charts, where a programming guideline causes performance loss, we denote the corresponding execution time with a bar of darker color.)

Nevertheless, we can set up some categories for the guidelines based on the scale of the resulting gain. *Global static data*, *Avoiding eval*, *Loop unrolling*, and *Local variables* are guidelines which result in a more than 50% runtime reduction in all browsers (Firefox being an exception for *Local variables*).

Another category is of those programming guidelines, which still bring gain but the speed-up is not so significant. *Common sub-expression elimination*, *Function inlining (c)*, and *Caching members* fall into this group (Firefox being an exception again, now for *Caching members*).

The final category consists of *Creating objects* and *Function inlining (b)*, where no clear suggestion can be made. Either leaving the code as is or applying a guideline can be a good or a bad choice, depending on the browser that will be used to

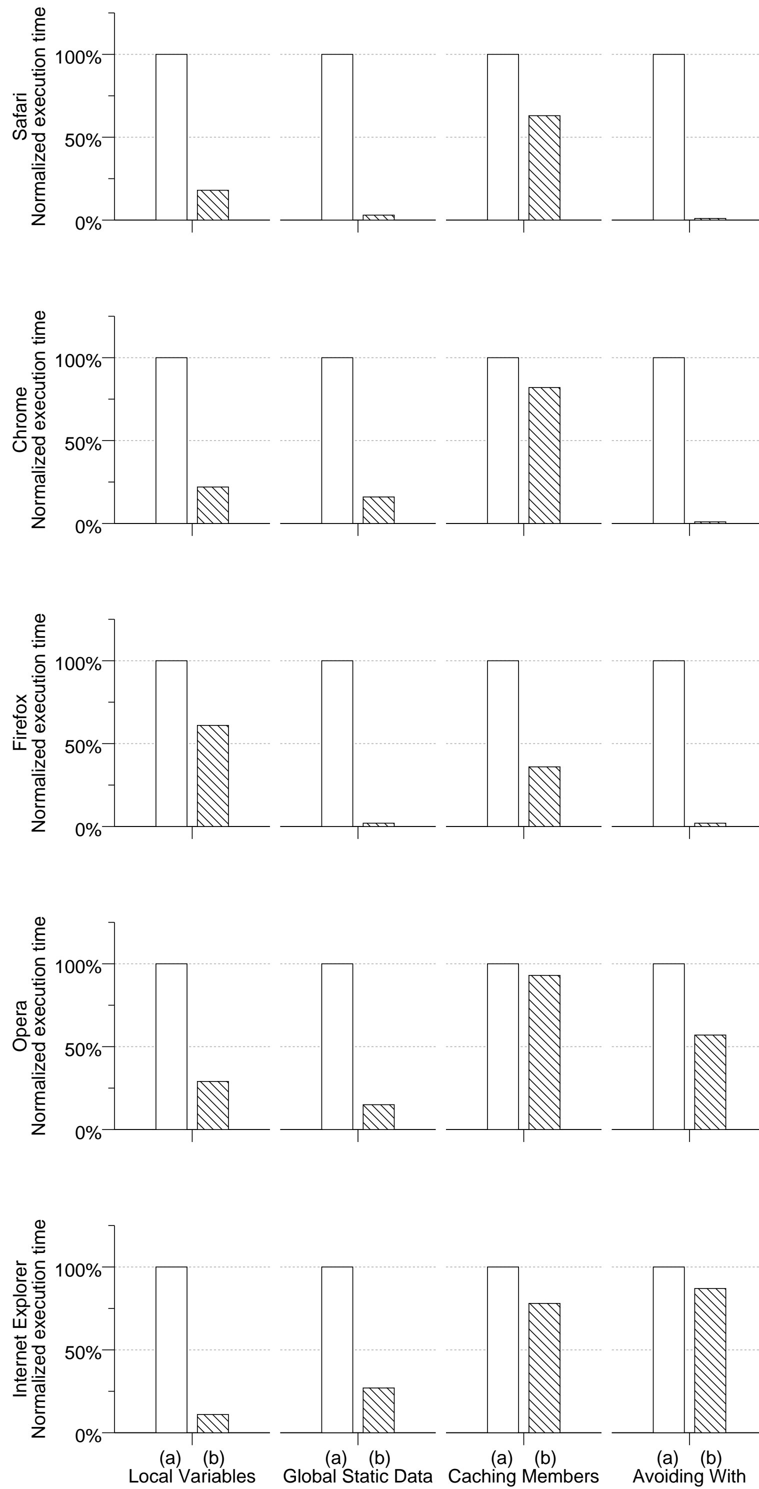

Fig. 13: The effect of the guidelines.

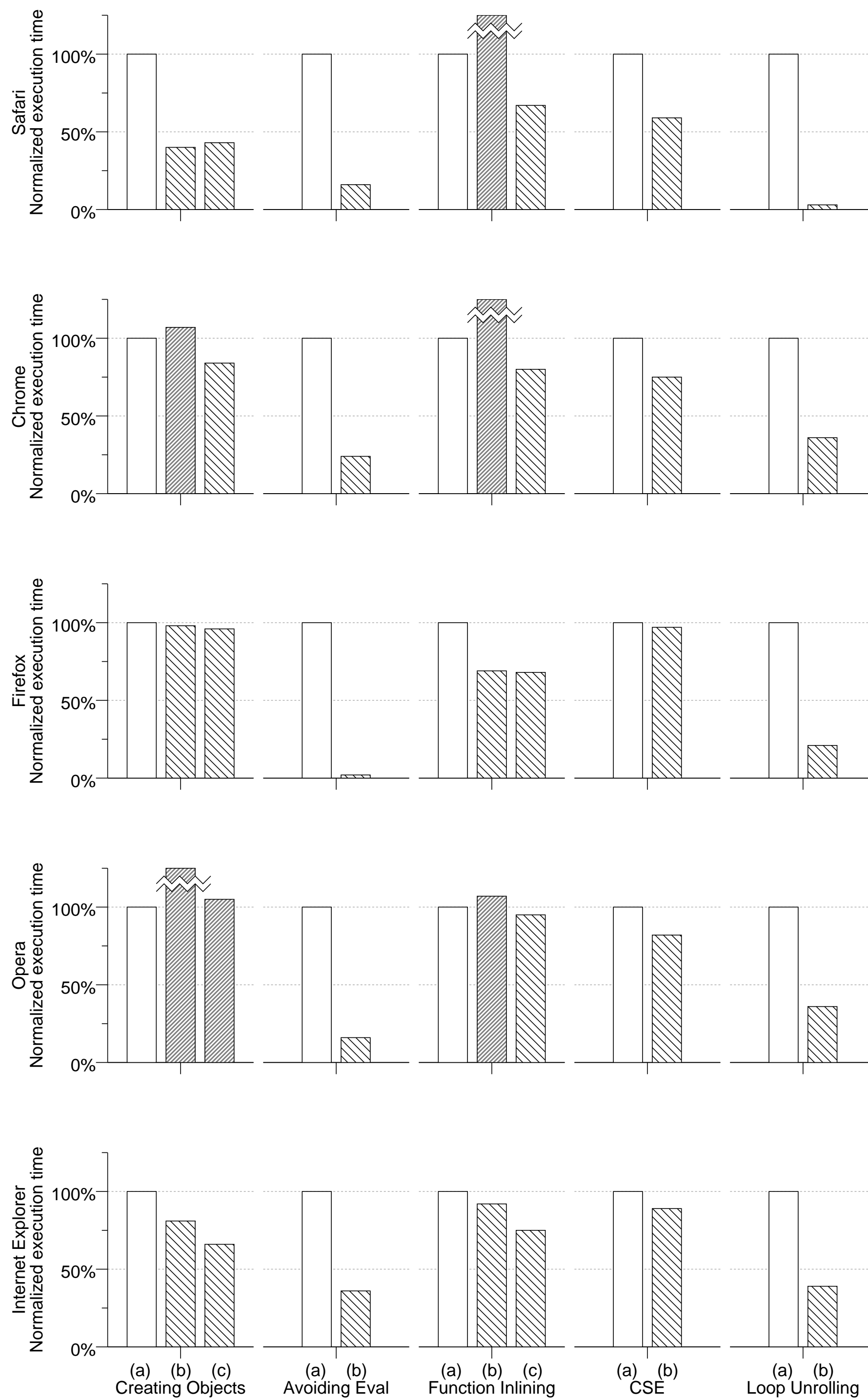

Fig. 14: The effect of the guidelines.

execute the code. (Nonetheless, for creating objects, the JSON notation might still be a valid approach, since in four browsers out of five, it speeds up the execution, and the performance loss is marginal for the remaining one.)

We have deliberately left out the *Avoiding with* guideline from the above classification. It is interesting to see how different the effect of the guideline is depending on the execution model of the JavaScript engine of the browser. While in those browsers where JIT technology is used (Safari, Chrome, and Firefox), the effect of the guideline is huge, in the case of interpreters (Opera and Internet Explorer) it is not that significant. (*Global static data* and *Caching members* show a bit similar tendency but because of Chrome, the results are ambiguous in those cases.)

Here, we have to refer to our previous work [Herczeg *et al.* 2009]. In that paper, we experimented with WebKit, which is the basis of the here-used Safari web browser. WebKit is a good subject for experimentation, since it contains both a JavaScript interpreter and a JIT compiler. We executed the same examples as we presented in Section 2 in both execution models and made some interesting observations. One of the observations was that all programs that conformed to the guidelines were faster than their unoptimized counterparts, irrespectively of the execution model. Another observation of ours was that the effect of the guidelines was in almost all cases higher in JIT execution model than in interpreted mode.

In the light of our multi-browser experiments discussed above, we can state that the observations do not change substantially. If we omit the (b) options of object creation and function calls, we can state that *almost all programs that conform to the guidelines are faster on all widespread browsers than their unoptimized counterparts, irrespectively of the execution model* (JSON object creation in Opera being the only weak exception). Moreover, based on the results of *Avoiding with* and partially on the results of *Global static data* and *Caching members* also, we can also state that *often the effect of the guidelines on performance can be higher in JIT execution model than in an interpreter.*

Thus, the conclusions remain valid as well. Performance guidelines are still useful and valid for interpreter and JIT-based JavaScript engines alike. Thus, programmers should not leave all the optimization work to JavaScript engines. Moreover, it seems that the importance of the guidelines does not decrease with the introduction of JIT engines but it might even increase!

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 [Bakota *et al.* 2008]. 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 [Gyimóthy *et al.* 2005]. This revelation leads many project owners to publish coding guidelines regardless of the language. Hence, guidelines are available for Java [Geotechnical Software Services 2008], PHP [Waring 2003], Action Script [Terracini 2007] 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 [Greenberg 2001][Garcia 2005][Zakas 2009]. 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 [Muchnick 1997][Nielson *et al.* 1999][Morgan 1998][Cooper and Torczon 2004][Allen and Kennedy 2002].

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 [Mozilla Foundation 2008]. 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 [Chang *et al.* 2009][Gal *et al.* 2009]. 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 [Google, Inc. 2008] and Apple's SquirrelFish Extreme [Stachowiak 2008] engines. Although the generated code is not adaptive, the fast paths in the generated code are based on static profiles.

6. Summary

In this paper, we investigated whether guidelines for JavaScript programs are still necessary and valid? This is a timely question, since in these days, new JavaScript engines are introduced in most of the popular web browsers, which apply state-of-the-art just-in-time compilation technology. However, it is important to know whether guidelines compiled for older 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 five widespread web browsers, some of them being still interpreter-based and some already applying JIT technology. Our investigations have led to the following observations (which underpin some earlier observations of ours):

- Generally, programs that conform to the guidelines are faster on all widespread browsers than their unoptimized counterparts, irrespectively of the execution model of the browser.
- Often, the effect of the guidelines on performance can be higher in JIT execution model than in an interpreter.

Thus, we can conclude that performance guidelines still seem to be valid, for interpreter and JIT-based JavaScript engines alike. 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.

There is still work to be done on the field of guidelines. In our current experiments, we investigated artificial example programs only. In future research, we plan to repeat the experiments on *de facto* standard JavaScript benchmark suites, like SunSpider [Apple, Inc. 2007] or the V8 Benchmark Suite [Google, Inc. 2009],

and on our own compilation, codenamed WindScorpion [University of Szeged 2008], thus measuring the effect of coding guidelines on real-life (or close to real-life) applications.

Acknowledgements

This research was partially supported by the TÁMOP-4.2.2/08/1/2008-0008 program of the Hungarian National Development Agency.

References

Allen, Randy and Kennedy, Ken. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann.

Apple, Inc. 2007. SunSpider JavaScript benchmark. `http://www2.webkit.org/perf/sunspider-0.9/sunspider.html` (Accessed 3 October 2010).

Bakota, Tibor, Beszédes, Árpád, Ferenc, Rudolf, and Gyimóthy, Tibor. 2008. Continuous Software Quality Supervision Using SourceInventory and Columbus. In *Research Demonstrations of 30th International Conference on Software Engineering (ICSE'08)*.

Chang, Mason, Smith, Edwin, Reitmaier, Rick, Bebenita, Michael, Gal, Andreas, Wimmer, Christian, Eich, Brendan, and Franz, Michael. 2009. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE 2009)*. Washington, DC, USA, 71–80.

Cooper, Keith D. and Torczon, Linda. 2004. *Engineering a Compiler*. Morgan Kaufmann.

Gal, Andreas, Eich, Brendan, Shaver, Mike, Anderson, David, Mandelin, David, Haghighat, Mohammad R., Kaplan, Blake, Hoare, Graydon, Zbarsky, Boris, Orendorff, Jason, Ruderman, Jesse, Smith, Edwin, Reitmaier, Rick, Bebenita, Michael, Chang, Mason, and Franz, Michael. 2009. Trace-based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI'09)*. Dublin, Ireland, 465–478.

Garcia, Edel. 2005. JavaScript optimization FAQs. `http://www.miislita.com/searchito/javascript-optimization.html` (Accessed 3 October 2010).

Geotechnical Software Services. 2008. Java Programming Style Guidelines – version 6.1. `http://geosoft.no/development/javastyle.html` (Accessed 1 July 2009).

Google, Inc. 2006. Google Web Toolkit. `http://code.google.com/webtoolkit/` (Accessed 7 November 2009).

Google, Inc. 2008. Google V8 Engine. `http://code.google.com/p/v8/` (Accessed 7 November 2009).

Google, Inc. 2009. V8 Benchmark Suite – version 5. `http://v8.googlecode.com/svn/data/benchmarks/v5/run.html` (Accessed 3 October 2010).

Greenberg, Jeff. 2001. JavaScript optimization. `http://home.earthlink.net/~kendrasg/info/js_opt/` (Accessed 7 November 2009).

Gyimóthy, Tibor, Ferenc, Rudolf, and Siket, István. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. In *IEEE Transactions on Software Engineering*, Volume 31. IEEE Computer Society, 897–910.

HERCZEG, ZOLTÁN, LÓKI, GÁBOR, SZIRBUCZ, TAMÁS, AND KISS, ÁKOS. 2009. Guidelines for JavaScript Programs: Are They Still Necessary? In *Proceedings of the 11th Symposium on Programming Languages and Software Tools (SPLST'09) and 7th Nordic Workshop on Model Driven Software Engineering (NW-MODE'09)*. Tampere University of Technology, Tampere, Finland, 59–71.

KRALL, ANDREAS. 1998. Efficient JavaVM Just-in-Time Compilation. In *Proceedings of the 1998 International Conference of Parallel Architectures and Compilation Techniques (PACT'98)*. Paris, France, 205–212.

LINDSTRÖM, JENS. 2009. Carakan. `http://my.opera.com/core/blog/2009/02/04/carakan` (Accessed 9 November 2009).

LOBAN, SCOTT. 2001. Language Choices for Client-Side Scripting. `http://www.peachpit.com/articles/article.aspx?p=24266` (Accessed 1 July 2009).

MORGAN, ROBERT. 1998. *Building an Optimizing Compiler*. Digital Press.

MOZILLA FOUNDATION. 2008. TraceMonkey. `https://wiki.mozilla.org/JavaScript:TraceMonkey` (Accessed 1 July 2009).

MUCHNICK, STEVEN S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

NIELSON, FLEMMING, NIELSON, HANNE RIIS, AND HANKIN, CHRIS. 1999. *Principles of Program Analysis*. Springer.

NIYOGI, SHANKU. 2010. The New JavaScript Engine in Internet Explorer 9. `http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx` (Accessed 30 October 2010).

STACHOWIAK, MACIEJ. 2008. Introducing SquirrelFish Extreme. `http://webkit.org/blog/214/introducing-squirrelfish-extreme/` (Accessed 1 July 2009).

TERRACINI, FABIO. 2007. Adobe Flex Coding Guidelines – version 1.2. `http://blog.dclick.com.br/wp-content/uploads/adobe-flex-coding-guidelines-v12-english.pdf` (Accessed 1 July 2009).

UEBERHUBER, CHRISTOPH W. 1997. *Numerical computation: methods, software, and analysis*. Springer.

UNIVERSITY OF SZEGED. 2008. WindScorpion. `http://www.sed.hu/webkit/?page=downloads` (Accessed 3 October 2010).

WARING, PAUL. 2003. PHP coding guidelines. `http://www.evolt.org/node/60247` (Accessed 1 July 2009).

ZAKAS, NICHOLAS C. 2009. Speed up your JavaScript: The talk. `http://www.nczonline.net/blog/2009/06/05/speed-up-your-javascript-the-talk/` (Accessed 1 July 2009).