Applying meta-functions for improving JavaScript code performance

Ricardo Medel, Alexis Ferreyra, Nestor Navaro and Emanuel Ravera
Departamento de Ingeniería en Sistemas de Información
Universidad Tecnológica Nacional - Facultad Regional Córdoba
Córdoba, Argentina
Email: ricardo.h.medel@gmail.com, alexis.ferreyra@gmail.com, nestornav@gmail.com, ravera.emanuel@gmail.com

***Abstract*—In recent years, the expansion of the World Wide Web, and web runtimes in particular, to all kind of devices has rendered the JavaScript performance in a hot topic. Several approaches to improve the performance of JavaScript applications have been tried by the industrial and research communities. In this paper we review the most popular approaches and propose a novel solution based on meta-programming and source code rewriting. The preliminary results of our experiments are very promising, although more studies are required to know to what extent this approach can improve the performance of real-life JavaScript programs.**

***Index Terms*—Performance, Meta-programming, Macros, Rewriting, JavaScript, Compiler**

I. Introduction

Improving JavaScript performance have been a hot topic in recent years. The ubiquity of the World Wide Web, and web runtimes in particular, across all kind of devices in addition to the increasing complexity of applications written in JavaScript have drawn attention of the industrial and research communities towards the improvement of JavaScript performance and security.

In this paper we focus on the performance aspects of this problem. Here we propose a novel solution based on meta-programming and code rewriting, show some preliminary results based on a small set of experiments, and then review other approaches and compare them with our proposal.

Most of the work to boost the performance of programs written in JavaScript is focused on improving the compiler/execution infrastructure. Such approach is implemented by the Safari SquirrelFish runtime [10], the just in time (JIT) compiler Google V8 [1], and applied on the trace based optimizations [2] by the Mozilla team. A different approach, also tried by Mozilla, is asm.js [3], a subset of JavaScript from which highly efficient code can be generated on the fly. Intel improves the performance by adding new extensions to the language, like SIMD [4] instructions or support for parallel execution [5] of certain APIs.

Our work, on the other hand, explores the feasibility of improving performance by using source code rewriting techniques at compile time, well before the source code reaches the JavaScript runtime. We found several cases where JavaScript performance is highly sensitive to the way in which the programmer writes the code. For example, as we show in Section II, using the `for-in` statement to iterate an array object is at least 25 times slower than using a standard C-style `for` to iterate the same array. Even more remarkable, different APIs generating the same, or semantically similar, results can show a big difference in execution times. For example, using the API `document.getElementByTagName` to find elements in the DOM can be 200 times faster than using the API `document.querySelectorAll`.

Therefore, we propose to improve the performance of JavaScript programs by analysing this kind of cases, which are out of reach of JIT compilers and runtimes. To test our hypothesis we initially identified a number of patterns amenable of being optimized and then implemented a set of simple experiments to check the performance gains. Once we showed that replacing slow patterns with the fast ones provided measurable performance benefits, we designed and implemented a general tool to easily automate this refactoring. The tool, named PumaScript, is a JavaScript dialect extended with program introspection and rewriting capabilities. Finally, by using PumaScript we are able of implementing meta-functions to automatically rewrite slow code with faster, semantically equivalent code.

In the following section we describe in detail the patterns we discovered and the code that can replace them. Next, we explain the implementation of PumaScript and how to use meta-functions to rewrite slow code. In the fourth section, we review different approaches to solve the same problem, showing their advantages and disadvantages. Finally, we close this paper with an analysis of our results and proposing future lines of work.

II. Analysis of JavaScript Patterns

In this section we compare the performance of several code patterns with semantically similar code but with different execution time. We also analyze and propose solutions for the cases where the semantics is not exactly the same.

We identified five JavaScript patterns showed in Table I. The second column of the table describes the pattern, while the third column shows first the original, slower code, and then the faster, semantically equivalent code.

In order to measure the different running times, we created and ran simple programs for each pattern. The observed performance improvements for the five patterns are shown in Table II. Each test was executed using two different desktop

1	Native selector by ID vs jQuery ID selector	Original Code `$("#test");`
		Improved Code `$(document.getElementById("test"));`
2	Iterate an array using `for in` vs C `for` statements	Original Code `var array = [1,2,3 ...]; for (var i in array) { array[i] +=1; }`
		Improved Code `var array = [1,2,3 ...]; for (var i=0; i<array.length; i++) { array[i] +=1; }`
3	Round an integer using parseInt vs bitwise operator	Original Code `var number = Math.random() * 1000; parseInt(number);`
		Improved Code `var number = Math.random() * 1000; number | 0;`
4	querySelectorAll vs getElementsByClassName	Original Code `var items = document.querySelectorAll(".test");`
		Improved Code `var item = document.getElementsByClassName("test");`
5	querySelectorAll vs getElementsByTagName	Original Code `var items = document.querySelectorAll("test");`
		Improved Code `var items = document.getElementsByTagName("test");`

TABLE I
JAVASCRIPT PATTERNS IN THEIR ORIGINAL AND IMPROVED FORM.

	PC Chrome v36	PC Mozilla v30	Android Tablet Native Browser	Android Tablet Chrome v36
1	2.15x	1.49x	1.98x	1.79x
2	55.54x	167.60x	27.96x	25.26x
3	12.91x	33.30x	11.24x	4.75x
4	138.88x	364.97x	915.23x	394.26x
5	236.16x	393.29x	213.69x	146.89x

TABLE II
IMPROVEMENT RATE FOR EACH PATTERN COMPARING SLOW AND FAST CODE.

browsers and two different Android browsers. The hardware used to run on desktop browsers was a notebook Lenovo T430, 4GB RAM, processor Intel Core i5-3320M 2.60Hz, with Windows 8 operating system. For benchmarking Android browsers we used a Tablet Asus MeMO Pad 7 with an Intel Atom Z3560 1.83GHz quad core processor, 2GB RAM and Android 4.4.2.

As it can be seen in the table, the improvement in performance between the slower and the fastest version of a pattern ranges between 1.49 times for the worst case (pattern 1 on a desktop PC with a Mozilla browser) and up to 915.23 times for the best case (pattern 4 on an Android tablet).

A. Solving Semantic Differences

Notice, however, that patterns 4 and 5 are optimistic transformations, since the original and improved code will generate the same result most of the time, but not always. This discrepancy exists because the `querySelectorAll` method returns an instance of a *NodeList* object, while `getElementsByClassName` and `getElementsByTagName` return an *HTMLCollection* object. Both objects, *NodeList* and *HTMLCollection* are similar, since both are collections containing the properties *length* and *item()*, which are used to get the number of items in the collection and to get an specific item, respectively. But because both collections use different constructors, a code that checks for the instance type of the collection can fail in the translated version.

Another consideration to take in account is that *HTMLCollection* objects are live collections, that is, when the DOM is updated the collection is updated. For example, after retrieving all nodes with class name *class1* by using the `getElementsByClassName` method, if a new node with *class1* class name is added to the DOM, the collection will be automatically updated.

Although for most cases the differences between *HTMLCollection* and *NodeList* will not change the semantic of

```
function createNodeList (elements) {
    var fragment = document.createDocumentFragment();
    for(var i; i < elements.length; i++)
        fragment.appendChild(elements[i]);
    return fragment.childNodes;
};
```

Fig. 1. Function that converts an *HTMLCollection* into a *NodeList*.

the program, it may be the case. Fortunately, there is a way to circumvent this semantic difference between the two collection types. When rewriting calls to `querySelectorAll` into calls to `getElementsByClassName` or `getElementsByTagName`, it is possible to wrap the returned collection with a new *NodeList* collection. Figure 1 shows the method `createNodeList`, which converts an *HTMLCollection* object into a *NodeList* collection.

After applying these changes for patterns 4 and 5 and running the benchmarks again, it was found that the improvement in a PC using Chrome was down from 138.88 times faster to a more conservative 17.2 times faster. Still, there is a sizable improvement in performance when using `getElementsByClassName` or `getElementsByTagName` methods instead of `querySelectorAll`.

III. Implementing a Meta-Programming and Rewriting Infrastructure

In order to automate the process of rewriting the identified patterns, we designed a new JavaScript dialect and implemented a rewriting infrastructure. The resulting framework, named PumaScript, provides a general tool to experiment with code introspection and meta-programming applied to improving language performance.

PumaScript main feature is the support for meta-functions, a mechanism similar to the programmable macro-expansion systems available in Lisp, Ruby, and other programming languages. Like other macro systems, PumaScript meta-functions can expand caller expressions inline. When called, a meta-function takes the decorated Abstract Syntax Tree (AST) of the arguments and returns an AST to be used as a replacement for the caller expression.

However, there are two key differences between PumaScript meta-functions and other macro systems:

1) Under certain conditions, a PumaScript meta-function can choose not to expand a certain occurrence of a caller expression, returning a *null* value instead of an AST.
2) PumaScript does not have a special *macro-expansion* phase before fully executing the program. Instead, meta-functions are live functions just like normal functions and can be called any time during the lifetime of the program.

PumaScript meta-functions can execute any arbitrary computation, including calling other normal functions or meta-functions. Additionally, all meta-functions have access to special *intrinsic functions* which provide access to the program AST for introspection and re-writing of any portion of it.

```
/* @meta */
function sum(a, b){
    return pumaAst( $a +  $b);
}

/* The following call to sum(5, 6),
will expand to the 5 + 6 expression */

sum(5, 6);
```

Fig. 2. A simple meta-function and its invocation.

As a simple example of a PumaScript meta-function, the Figure 2 shows a meta-function that rewrites the call into an addition expression.

The current high level implementation and execution process of PumaScript is shown in Figure 3. We use the Esprima library [6] to parse the JavaScript-like syntax. Then the PumaScript runtime is used to execute the Abstract Syntax Tree following the standard JavaScript semantic plus the additional rules and semantics added by PumaScript. Once the program is executed, PumaScript runtime discards the meta-functions nodes and the resulting Decorated Syntax Tree is processed by the Escodegen library [7] in order to pretty print the program into standard JavaScript.

A. PumaScript Meta-Functions

As seen in Figure 2, PumaScript meta-functions are written just like normal JavaScript functions, with the annotation `@meta` in a comment before the function declaration. This method is used to avoid introducing a new syntax requirement, making PumaScript backward compatible with standard JavaScript.

Meta-functions work in a similar way than regular JavaScript functions, with three specific differences:

1) All its parameters will evaluate to a reference into the caller argument Decorated Syntax Tree at the moment of execution. For example, when calling the meta-function `foo(a,b)` with actual argument expressions `(2 * x)` and `(3 * y)`, the parameter `a` will take the value of the syntax tree for `(2 * x)` and the parameter `b` will take the value of the syntax tree `(3 * y)`.
2) It must return a valid syntax tree or *null*. If the return value is *null*, then the caller expression will not be rewritten. Otherwise, if the return value is a non-null syntax tree, the caller expression will be replaced with the returned syntax tree, actually rewriting the caller expression in the process.
3) It has access to a reserved context with intrinsic objects and functions that can be used to make introspection into the program or to rewrite any portion of the program. Sample intrinsic objects and functions available in the meta-function context are: `pumaAst`, `context`, and `pumaFindByType`.

In the example of Figure 4 the meta-function `firstLetter` rewrites its caller only if the actual argument is a string literal. In the second call the function returns *null* and then the caller is not rewritten.

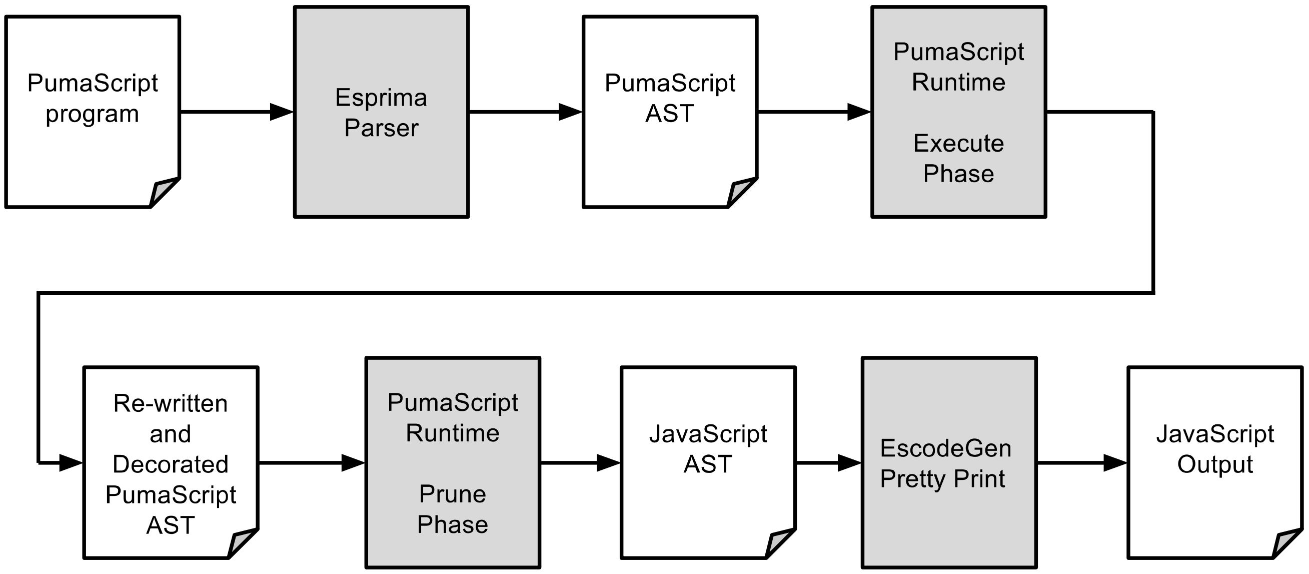
PumaScript program
Esprima Parser
PumaScript AST
PumaScript Runtime Execute Phase
Re-written and Decorated PumaScript AST
PumaScript Runtime Prune Phase
JavaScript AST
EscodeGen Pretty Print
JavaScript Output

Fig. 3. PumaScript program execution workflow and high level modules.

```
// Program sent to PumaScript

/* @meta */
function firstLetter(valueExp){
    var ast = null;
    if(valueExp.type === "Literal"){
        ast = valueExp;
        ast.value = ast.value.substring(0, 1);
    }
    return ast;
}

// This call will be rewritten to "H";
firstLetter("Hello World");

// This call will not be rewritten because
// the argument expression is not a literal
firstLetter("Hello " + "World");

// Output of PumaScript
"H";
firstLetter("Hello " + "World");
```

Fig. 4. A meta-function replacing string literals and its invocations.

As an example of use of intrinsic objects and functions, Figure 5 shows a PumaScript meta-function that counts all the occurrences of `for` statements in a script and shows the resulting number by using the standard console object. In this example we use the intrinsic function `pumaFindByType` and the intrinsic object *pumaProgram*. These objects and functions can be used to introspect any portion of the program, not only the current context that is calling the meta-function.

B. Rewriting Code

In this section we describe how automatic rewriting of the patterns presented in Table I was implemented using the PumaScript language. For clarity and space reasons, we only include the first two patterns shown in the mentioned Table.

```
/* @meta */
function countForStatemets() {
  var forStas = pumaFindByType(pumaProgram, "ForStatement");
  console.log("For statements found: " + forStas.length);
  return null;
}
```

Fig. 5. Meta-function counting the number of `for` statements in a program.

1) Pattern 1 - Rewrite jQuery Selector Calls:: Figure 6 shows a meta-function that rewrites jQuery selectors by Id into the more efficient JavaScript native API `document.getElementById`. Line 6 checks that the actual argument is a *Literal* node and tests for the regular expression used to match *selectors by Id*. By executing the `then` branch, it removes the # character from the beginning and returns a new abstract syntax tree (AST) using `document.getElementById` and the provided argument with the modified string literal. The intrinsic function `pumaAst` is used to build the returned AST, starting from a template where the local variables are expanded with their actual values.

The simplest use case happens when the meta-function is called with a simple string literal argument.

```
// Invocation with literal
var myElement = $("#Element_Id_1");
```

In this case, it is always safe to rewrite the invocation. Therefore, the execution of the meta-function of Figure 6 will follow the `then` branch of the `if-else` statement of line 6.

On the other hand, when the actual argument is not a

```
/* @meta */
function $(valueExp){
  var regex = /^#\b[a-zA-Z0-9_]+\b$/;
  var argValue = {};
  var substr = ’’;

  if(valueExp.type === "Literal" && regex.test(valueExp.value)){
    valueExp.value = valueExp.value.substring(1);
    return pumaAst($(document.getElementById($valueExp)));
  }
  else if(OPTIMISTIC_REWRITE){
    argValue = evalPumaAst(valueExp).value;
    if(regex.test(argValue)){
      console.log("WARNING: Optimistic rewrite at line("
      + valueExp.loc.start.line + ")");
      substr = valueExp.substring(1);
      return pumaAst($(document.getElementById($substr)));
    }
  }
  return null;
}
```

Fig. 6. Meta-function to rewrite jQuery selectors by Id.

simple literal, the meta-function uses the intrinsic function `evalPumaAst` to evaluate any portion of AST in the current execution context. In this case, it uses a flag variable `OPTIMISTIC_REWRITE` to enable optimistic rewriting when it can check that at least one execution of the caller expression matches the *selector by Id* form.

For example, the following invocation may not be safe to rewrite if the variable `element_id` is an argument into a function, but by using the `evalPumaAst` intrinsic function in line 11 (Fig. 6), the meta-function can detect that the call is safe to rewrite into a more efficient API call.

```
// Invocation with non-trivial expression
var element_id = "5";
var myOtherElement = $("#Element_Id_" + element_id);
```

Finally, there are cases where the selector does not match a simple *selector by Id.*

```
// Invocation that does not match a selector by Id
var myOtherClassElements = $(".Class_Id_" + element_id);
```

For this invocation, the meta-function is capable of identifying the problem easily: no matter what value the variable `element_id` takes, it will not form a valid *selector by Id.*

2) Pattern 2 - Rewrite `for-in` Statements:: In order to implement the automatic rewriting of `for-in` statements into C-style `for` statements, a different approach is needed. It is not possible to use a meta-function like a macro call which rewrites the caller expression. Instead, the meta-function to rewrite `for-in` statements needs to use intrinsic functions provided by the PumaScript runtime environment to introspect the AST of the running program.

Figure 7 shows the main meta-function used to rewrite `for-in` statements. In line 3, it uses the intrinsic function `pumaFindByType` and the intrinsic object `pumaProgram` to match all AST nodes whose type is *ForInStatement*. Then, the function iterates on this list and uses a helper function to rewrite each `for-in` subtree in the program.

The helper function to rewrite a single `for-in` statement, shown in Figure 8, has four main steps:

```
/* @meta */
function rewriteForIn() {
  var forIns = pumaFindByType(pumaProgram, "ForInStatement");
  console.log("For In statements found: " + forIns.length);

  for(var i = 0; i < forIns.length; i++) {
    rewriteSingleForIn(forIns[i]);
  }
  return null;
}
```

Fig. 7. Main meta-function to rewrite all `for-in` statements.

```
function rewriteSingleForIn(forInAst){
  var left = forInAst.left;
  var right = forInAst.right;
  var itemName;
  var tempId;

  // Detect which kind of iteration variable it uses
  if (left.type === "Identifier") { itemName = left;}
  else if (left.type === "VariableDeclaration") {
    tempId = left.declarations[0].id;
    itemName = pumaAst( $tempId );}
  else {return;}

  // Prepare fallback version and optimized AST
  var cloneForIn = pumaCloneAst(forInAst);
  var optimizedFor = pumaCloneAst(forInAst);

  optimizedFor.type = "ForStatement";
  optimizedFor.init = left;
  optimizedFor.test = pumaAst($itemName < $right.length);
  optimizedFor.update = pumaAst($itemName = $itemName + 1);

  // Create type-guard to test runtime type
  var temp = pumaAst(function(){
    if (Array.isArray($right)) $optimizedFor;
    else $cloneForIn;});

  var tempIf = pumaFindByType( temp, "IfStatement")[0];

  // Replace original node with type-guarded one
  forInAst.type = tempIf.type;
  forInAst.test = tempIf.test;
  forInAst.consequent = tempIf.consequent;
  forInAst.alternate = tempIf.alternate;
}
```

Fig. 8. Helper function to rewrite a `for-in` statement.

1) It detects if the `for-in` statement uses a variable declaration as the iteration reference or an existing variable.
2) It creates a new AST for an equivalent C-style `for` statement.
3) It creates an `if` statement that will be used as type guard to fallback into the original `for-in` if the type of the collection expression to be iterated is not an array.
4) It replaces the original `for-in` AST node with the generated `if` AST node.

Note that the function in Figure 8 is not marked as a meta-function, as PumaScript can arbitrarily mix meta-functions with normal functions.

Figure 9 shows an example of a client program that calls the meta-function to rewrite all the program’s `for-in` statements. The output obtained from running the program is shown in Figure 10.

```
var array = [1,2,3,4,5,6,7,8,9,0];
var i = 0;

// Test for-in with existing iteration variable
for(i in array){ array[i] += 1;}

// Test for-in with new iteration variable
for(var j in array){ array[j] += 1;}

// Call to meta-function to optimize for-in
rewriteForIn();
```

Fig. 9. Sample client program which uses the *rewriteForIn* meta-function.

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
var i = 0;

if (Array.isArray(array))
  for (i; i < array.length; i = i + 1) { array[i] += 1;}
else
  for (i in array) { array[i] += 1;}

if (Array.isArray(array))
  for (var j; j < array.length; j = j + 1) {
    array[j] += 1; }
else
  for (var j in array) { array[j] += 1;}
```

Fig. 10. Output generated by PumaScript after running the program of Figure 9.

IV. Related Work

To our knowledge, source-to-source code rewriting techniques were never before used to improve JavaScript performance. However, other approaches have been developed previously to improve the performance of JavaScript programs. In this section we review other source code rewriting tools, describe commonly used JavaScript pre-processing tools, and analyze the most relevant approaches to compare them with our proposal.

A. Rewriting and Pre-processing Tools

1) Source Code Rewriting Frameworks:: A number of existing frameworks and tool chains can be used to implement end-to-end source code cross-compilation or refactoring. Stratego XT [8] and DMS [9] are two of the more mature frameworks to build source-to-source transformation tools applying rewriting techniques. The main strength of these frameworks is their flexibility, as they provide end-to-end tools to build parsers, rewriting scripts, semantic analyzers, and pretty printers. Our approach, as implemented by PumaScript, is simpler and does not aim to be a generic tool to transform from any source to any possible target. It is focused in JavaScript language rewriting.

Moreover, building and end-to-end JavaScript-to-JavaScript tool with these frameworks requires an important amount of work because every major component must be created using these tools. In contrast, in the implementation of PumaScript we reuse existing and tested components like the Esprima parser [6] for the front-end and EscodeGen [7] for pretty printing.

Another major difference between these frameworks and our solution, is that to implement transformations in these frameworks the developer must learn specific languages based on the tree rewriting paradigm. This programming paradigm is not well known by most developers. Instead, PumaScript meta-functions use the same JavaScript programming language that any JavaScript developer already knows. Our approach does not introduce a significantly new programming paradigm beyond requiring familiarity with macro-expansion systems, which are already available in a number of well known programming languages.

2) Code Minifiers and Pre-Compilers:: Code minifiers have been utilized by the JavaScript community for a long time. Simple minifiers like JSMin [10] and JSZap [11] provide mostly bandwidth optimization but not measurable performance improvements. Elaborated precompilers, like Google Closure Compiler [12], are capable of more advanced optimizations like code inlining and removing unused variables. However, those optimizations are provided as a mean to shorten the source code and not as a way to improve performance.

B. JavaScript Performance Improvement Approaches

1) Runtimes and Just in Time Compilers:: The greatest performance improvement in JavaScript language has been related with the progression from using runtimes, like Safari SquirrelFish [13], to more advanced Just In Time (JIT) compilers, such as Chrome V8 [1], Mozilla SpiderMonkey [2], or Microsoft Chakra [14].

All of these JIT-based engines reuse techniques previously used in other language runtimes, most notably the Java HotSpot compiler [15]. Although the introduction of JIT compilers has provided an improvement of at least one order of magnitude, even the more advanced JIT engines cannot optimize certain language patterns, such as `for-in` vs C style `for`. Moreover, JIT engines are not good candidates to incorporate optimizations related to similar APIs with different performance, like the jQuery selectors vs native APIs cases identified in this work.

2) Language Extensions:: Language extensions like Mozilla asm.js [3] or Intel proposed SIMD [4] and parallel execution [5] extensions are capable of providing important performance benefits for certain use cases. However, these approaches suffer from several limitations. First, developers must embrace the language extensions by using them in their source code. Second, runtime providers must implement support for these extensions in their products. These limitations generate the classic chicken-and-egg problem: developers are not willing to invest effort in modifying their code until most runtime providers add support for a certain language extension, while at the same time, runtime providers are not encouraged to support the extensions because there are not enough projects using them.

In contrast, our approach can be used from day one by developers, not requiring them to change the source code and having immediate benefits in existing runtimes. The only additional cost for a developer to use our method is to add PumaScript and the rewriting scripts to the deployment process.

V. Summary and Future Work

In this work we show that the performance of JavaScript programs is highly sensitive to the use of a number of source code and API patterns. Therefore, sizable performance improvements can be achieved by replacing these patterns by semantically equivalent faster code.

Also, we introduce a JavaScript language extension and framework called PumaScript, which can be used to automate a number of source code rewriting tasks. This new framework adds meta-functions to JavaScript, allowing introspection and rewriting syntax trees on the fly, without requiring the program to be restarted or a specific macro-expansion phase included in the runtime.

Developers can integrate PumaScript rewriting infrastructure and transformation scripts into their continuous integration environments to optimize the source code before the deploying phase. Additionally, our novel approach to improve performance is complementary to progress in JavaScript runtimes, high performance language subsets and other efforts to improve the language performance.

Still, there is additional work to be done in order to validate how the exhibited performance benefits in our simple experiments translate to real life code. The first open task is to analyze how common these non-optimal patterns are in actual JavaScript source code. Moreover, it is possible that several other non-optimal patterns exist and can benefit from our approach.

Second, we have to use real life code from third parties to prove that the performance gain are similar to the shown in this work. We plan to use open source projects from public repositories to create a new, more comprehensive set of experiments.

Finally, we would like to explore the use of our PumaScript infrastructure in other applications related to JavaScript metaprogramming. For example, it could be applied to source code generation, construction of static, pre-compiled domain-specific languages, static and real-time source code analysis, application of aspect oriented programming, source code instrumentation, and source-to-source transformation to other scripting languages.

References

[1] Google Inc. V8 JavaScript virtual machine. https://github.com/v8/v8/wiki, 2017.

[2] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. R. Haghighat, M. Bebenita, M. Chang, and M Franz. Trace-based just-in-time type specialization for dynamic languages. In *Programming Language Design and Implementation (PLDI 2009)*, Dublin, Ireland, June 2009.

[3] David Herman, Luke Wagner, and Alon Zakai. asm.js specification. http://asmjs.org/spec/latest/, 2013.

[4] Intel Corporation. SIMD in JavaScript. https://01.org/node/1495.

[5] Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram. River trail: a path to parallelism in JavaScript. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, Indianapolis, IN, October 2013.

[6] ECMAScript parsing infrastructure for multipurpose analysis. http://esprima.org/.

[7] ECMAScript code generator EscodeGen. https://github.com/estools/escodegen.

[8] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1–2):52–70, June 2008.

[9] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *ICSE 04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Washington, DC, 2004. IEEE Computer Society.

[10] D. Crockford. JSMin: The JavaScript minifier. http://www.crockford.com/javascript/jsmin.html.

[11] Martin Burtscher, Benjamin Livshits, Benjamin G. Zorn, and Gaurav Sinha. JSZap: compressing JavaScript code. In *Proceedings of the 2010 USENIX conference on Web application development*, pages 4–4, Boston, MA, June 2010.

[12] Google Inc. Google closure compiler. https://developers.google.com/closure/compiler.

[13] Geoffrey Garen. Announcing SquirrelFish. https://www.webkit.org/blog/189/announcing-squirrelfish/, June 2008.

[14] Advances in JavaScript performance in IE10 and Windows 8. http://blogs.msdn.com/b/ie/archive/2012/06/13/advances-in-javascript-performance-in-ie10-and-windows-8.aspx.

[15] Michael Paleczny, Christopher Vick, and Cliff Click. The java hotspotTM server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–1, Monterey, CA, April 2001.