Photon: an open, extensible and native implementation of JavaScript

Erick Lavoie, Marc Feeley, Bruno Dufour
Université de Montréal
{lavoeric, feeley, dufour}@iro.umontreal.ca

Abstract

This is the text of the abstract.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—compilers, optimization, code generation, run-time environments

General Terms Algorithms, Performance, Design, Languages

Keywords JavaScript, virtual machine, compiler, self-hosted, optimization, implementation, bootstrap

1. Use Cases

1.1 Extending the object model

The next version of the JS standard, nicknamed *Harmony*, introduces native support for associative table, which they call `Map`. To avoid confusion with the meta-objects in Photon, we distinguish them by using an uppercase M for the associative table name and a lowercase m for the meta-objects.

Providing native support brings a performance advantage by allowing the implementation to heavily optimize the operations of the new object type. In this use case, we show that our system is both able to profit from a direct access to memory to implement native types and reuse much of the existing system to simplify the introduction of user-defined native types.

For simplicity of presentation, we use an implementation that performs a linear search for every Map operation. We compare an implementation done in regular JS with a native implementation done using extensions presented in previous sections. The regular JS implementation is given at figure 1. It uses two arrays to store keys and values. The native implementation is given at figure 2. It uses the payload section of the object to store the number of entries in the table and the entries themselves. Doing so eliminates a level of indirection. Comparing the time required to perform 2000 times each of the `set`,`get`,`has` and `delete` operations shows a 140% slowdown between the regular and the native implementation. The running times are given at table 1.

Additional considerations are required for a native implementation. First, extension of the object is necessary once the payload section is full. This is done by creating a clone of the current object with a bigger payload section and updating the extension slot of the original object. Every operation on the object then needs to retrieve the extended object before performing the actual operation. Second, the object protocol must be followed. In this case, a `__new_default__` method is necessary for correct behavior with the `new` operator. Third, implementation invariants need to be maintained. In this case, the inline cache implementation requires a common `map` for all objects created from the prototype. This is achieved by creating a new `map` and assigning it to the `__base_map__` property of the prototype object.

Implementation	Photon (s)	Slowdown
Regular JS Map	2.6	1.4
Extended JS Map	1.8	1.0

Table 1. Running times (in seconds) of different implementations of the Map object

Some things are provided for free. For example, the extensible object model provides the regular object behavior without requiring any additional code for the native implementation.

Comparing the two implementations shows that direct memory manipulation and additional considerations take twice the number of lines of code for an equivalent native implementation, to prove a 30% speedup.

1.2 Profiling the runtime behavior

The runtime profile of a JS program is currently laborious to obtain. The common approach is to instrument an existing implementation with various hooks to record runtime events. The major problem with this approach is that all implementations running in browsers have been heavily optimized for performance with no concern about instrumentation.

The existence of an object protocol allows a much easier way to instrument running programs. Additionally, the ability to dynamically modify the behavior of the system makes it possible to profile the running behavior for *a part* of the execution instead of having to choose between profiling for the whole execution or no profiling at all.

The following example shows how the number of regular objects created during execution can be obtained and how the instrumentation code can be removed once it is no longer needed. The example code is given at figure 3. Running this example shows that 3006 regular objects were created for performing the evaluation of $1 + 2$. After reverting the method, the original behavior is restored. All the instrumentation performed is compatible with the inline cache behavior. Note that responsability for avoiding recursion in the implemenation of object-protocol methods is left on the programmers shoulders. Knowledge of the code generated by the compiler is necessary for that purpose. In our opinion, the simplicity of the compilation model makes it reasonably straightforward.

SPLASH 2012 October 19-26, 2012, Tucson, Arizona, USA

```
function Map() {
    var that = this;
    that._keys = [];
    that._vals = [];

    return that;
}

(function () {
    function indexOf(keys, key) {
        var length = keys.length;
        for (var i = 0; i < length; ++i)
            if (keys[i] === key) return i;
        return -1;
    }

    Map.prototype.get = function (k) {
        var i = indexOf(this._keys, k);
        if (i >= 0) return this._vals[i];
        else return undefined;
    };

    Map.prototype.has = function (k) {
        return indexOf(this._keys, k) >= 0;
    };

    Map.prototype.set = function (k, v) {
        var i = indexOf(this._keys, k);
        if (i >= 0) return this._vals[i] = v;

        var keys = this._keys;
        var l    = keys.length;
        keys[l]  = k;
        this._vals[l] = v;
        return v;
    };

    Map.prototype.delete = function (k) {
        var i = indexOf(this._keys, k);
        if (i < 0) return false;

        var keys = this._keys;
        var last = keys.length-1;
        var vals = this._vals;

        if (i !== last) {
            keys[i] = keys[last];
            vals[i] = vals[last];
        }

        keys.length--;
        vals.length--;
        return true;
    };
})();
```

Figure 1. Regular JS Map implementation

1.3 Profiling compile-time information

Obtaining compile-time information is generally easier than runtime information. The common approach is to use or write a parser for the language and then reason on a defined representation. However some information, such as the number of a given operation generated by the compiler is compiler-dependent and harder to obtain since it depends on the optimizations performed.

Being able to intercede on a compiler's behavior permits the reuse of much of the infrastructure of the compiler to obtain compile-time information and it provides direct information on the actual compiler's behavior. The example given at figure 4 shows how to obtain the number and nature of message sends generated by the compiler for a given piece of code.

Again, since intercession on the compiler's behavior is done at runtime it is also possible to remove the cost of profiling when it is no longer needed without having to restart the program.

1.4 Changing the language being compiled

This use case explores the redefinition of a part of the compiler at runtime to support different syntaxes or semantics. The system allows completely replacing the existing compiler with a new one. For example, that could be used to provide better runtime performance for compiled code or to support a different language.

In this case, we chose to add support for a lisp dialect that uses most of the keywords and semantics of the JS language. Such an extension could allow support for syntax macros that would effectively bring syntactic extensibility to JS.

To keep reusing our beloved Fibonacci example again, an example of the syntax is given at figure 5. This is pretty much Scheme code with the exception that the `var` keyword is used to define global variables instead of `define` and the `function` keyword is used to define anonymous functions instead of `lambda`.

The implementation uses 114 lines of OMeta code to define a parser and scanner. Since the original parser is exposed on the global object, rebinding its value to the new parser effectively changes the syntax system-wide dynamically. After this change, the system can be serialized again to provide a stand-alone executable VM that understands a JS lisp-inspired dialect.

2. Empirical Evaluation

We perform an empirical evaluation of the resulting system to show that it is pratical as an exploration vehicle for implementation techniques. We first show that our current implementation is *simple* as mesured by the number of lines of code of the complete system. We then show that the implementation is *promising* in terms of performance.

2.1 Complexity

We use the total number of lines of code as a proxy for the complexity of the resulting system. To mitigate the effect of writing style, all the JS source code is subject to a source-to-source translation which removes all the comments and uniformize the writing style. The resulting output was chosen to approximate a hand-written coding style. For example, control statements always use curly-braces and `if` and `else` branches are written on different lines.

The system is written in OMeta, JS and C code. All the OMeta code is compiled to JS. The code required to compile the OMeta code is included in the total number of source lines of code (SLOC). The OMeta compiler is itself written in OMeta. The breakdown for SLOC is given at table 2. An interesting thing to notice is the effect of meta-circularity on the SLOC number. Since OMeta is meta-circular, the OMeta runtime used to generate the Photon Compiler is the same as the one used by the Photon Compiler to compile JS code. Similarly, the JS runtime used by the Photon Compiler is the same as the one used by JS code to execute. That code would have been duplicated if those runtimes had not been shared. A second interesting thing to notice is the fact that almost a fourth of the SLOC are written in C. Of those, two-thirds are runtime methods for objects and the rest concern bootstrap, serialization and initialization tasks. In the future, we expect to reduce the amount of C code as more of the runtime functionalities will be expressed only as JS code.

We used OMeta code to drastically reduce the number of lines of code to write and modify. As shown in table 3, for the Photon Compiler we gain a factor of 6 in expressivity compared to having written the same OMeta code by hand. The main gain was obtained through the concise notation for pattern-matching on arrays afforded by OMeta.

Since we needed extensions to JS for writing some of the VM components, it begs the question of what proportion of the VM was written using only regular JS code. This figure is interesting

Category	SLOC	%
OMeta Compiler(OMeta)	500	
OMeta Runtime (JS)	1000	
Photon Compiler (OMeta+JS)	5000	
Photon Optimizer (OMeta+JS)	500	
Runtime (JS)	3200	
Runtime (C)	2000	
Bootstrap, Serialization, etc. (C)	1000	
V8 Integration (C)	200	
Total	13400	100

Table 2. Source lines of code breakdown by component

Category	OMeta SLOC	JS SLOC	Expansion factor
OMeta Compiler	500	5000	10.0
Photon Compiler	1100	7200	6.5
Photon Optimizer	10	60	6.0
Total	1610	12260	7.6

Table 3. Expansion factor for OMeta code

because it gives a rough approximation of the amount of JS code that can be reused should the object representation or the extension syntax changes. If we also count the OMeta code, also compiled to regular JS, table 4 shows that 65% of the total code could be reused should the internal design of the VM changes. Let's note however that assumptions *semantically* encoded in regular JS code are not apparent here but we can say from experience that these represent a small part in terms of SLOC.

Category	SLOC	%
OMeta	1600	12
Regular JS	7100	53
Extended JS	1500	11
C	3200	24
Total	13400	100

Table 4. Proportion source lines of code by language

Comparison to one of our previous attempts at a JavaScript meta-circular VM[1] shows a reduction of a factor of 5 in terms of lines of code (13.4KLOC vs 75KLOC). The reduction in the SLOC number can mostly be attributed to the usage of OMeta as well as the elimination of the SSA representation and register allocator. Comparison to V8 shows a factor of 28 (13.4KLOC vs 375KLOC) although to be fair, V8 provides a complete support for JS and backends for x86 32-bit and 64-bit as well as ARM. Using the SLOC number as a proxy shows that Photon is a much *simpler* system than other alternatives.

2.2 Performance

Performance was not the focus of the current implementation. We still compare our system to a state-of-the-art implementation to provide ballpark figures of its current performance. We show execution times for common sunspider benchmarks as well as v8-hosted and self-hosted compilation time. We then give explanations for the current bottlenecks and we propose implementation strategies to remove them.

All the numbers were obtained on an early 2011 MacBook Pro running OS X Lion 10.7.3 with a 2.2 GHz Intel Core i7 and 8 GB of memory. Both our system and d8 were compiled for x86 32-bit. We used revision 7928 of d8 and the 0.9.1 sunspider benchmarks.

Since our system does not support equality comparison (==), those were converted to identity comparisons (===). Code executing in the global environment was also put in a function to permit serialization of the compiled code when running inside Photon. It was done to avoid compilation when running tests. The same code is used for tests both on V8 and Photon. Those two modifications did not significantly affect the running time on V8.

2.2.1 Execution Time

We show some sunspider benchmarks results. As a reminder, we do not optimize constant arithmetic operations, every local variable is accessed on the stack and we follow the 32-bit calling conventions of C. Given these, the results look promising, since there is a lot of room for improvement using known techniques.

Two variations of some tests are introduced to illustrate the current bottlenecks of Photon. In the first one, we removed the string creation in the access-fannkuch.js benchmark. The resulting string was not used in the benchmarks, therefore it does not change the algorithm. We label the modified benchmark with (no string). In the second one, we replace the new operation in access-binary-tree.js with an internal cloning operation of a representant object with all the expected properties. We label the modified benchmark with (cloning). This benchmark cannot be run on V8 anymore, we therefore compare the running time on Photon with the original running time on V8.

Table 5 shows the relative speed of Photon and V8. We compare the speed of our system using inline caches (ic) to the speed of V8. This simple implementation strategy brings our system within a factor of 3 of V8 on a function call intensive benchmark such as controlflow-recursive.js. Our system is within a factor of 14 of V8 if we do not consider the two degenerated cases.

Test Name	Photon (ic)	V8	Slowdown
controlflow-recursive.js	1.845	0.584	3.16
access-nsieve.js	3.427	0.542	6.32
access-fannkuch.js	89.417	1.835	48.72
access-fannkuch.js (no string)	18.389	1.804	10.19
access-binary-tree.js	58.940	0.360	163.72
access-binary-tree.js (cloning)	5.000	-	13.88

Table 5. Execution time for 400 iterations of some sunspider benchmarks (seconds)

The two slowest running times can be explained as follow. During the original fannkuch benchmark, a linear search is performed for string internalization each time a string is created, making the running time proportional to the number of strings in the system. During the original binary tree benchmark, the creation of an object within Photon performs an introspective search on the prototype to find an existing base map as a JS property to maintain the inline cache invariants. Potential solutions to these two problems will be addressed in the discussion section.

Table 6 shows the effect of inline caches on execution time. Given the pervasive use of message sends in the object model implementation to late-bind object behavior, it is of little surprise that inline caches (ic) give a huge boost to performance. When not using them (noic) the system is between 4 and 51 times slower.

Test Name	Photon (noic)	Photon (ic)	Slowdown
controlflow-recursive.js	4.606	0.195	23.62
access-nsieve.js	17.716	0.346	51.20
access-fannkuch.js	84.504	9.079	9.30
access-fannkuch.js (no string)	78.079	1.835	42.55
access-binary-tree.js	23.428	5.973	3.92
access-binary-tree.js (cloning)	15.613	0.507	30.79

Table 6. Execution time for 40 iterations of some sunspider benchmarks (seconds)

2.2.2 Compilation Time

Table 7 gives the compilation time for two of the precedent benchmarks. It shows a factor of 100 between the self-hosted and the v8-hosted version. From the result given in the previous section, we believe this is mostly caused by the speed of the object creation protocol.

Additionally, bootstrapping the current system on V8 currently takes around 30 seconds. Given the slowdown factor when performing self-hosted compilation, we did not attempt a self-hosted bootstrap.

Test Name	Photon (ic)	Photon (noic)	Photon/V8
controlflow-recursive.js	10.780	8.858	0.085
access-nsieve.js	10.022	8.103	0.078
Total			

Table 7. Compilation Time for some benchmarks (seconds)

Table 8 gives the relative compilation time for each compilation phase. When v8-hosted compilation is done, parsing dominates the compilation time. However, when self-hosted compilation is performed, code generation is dominating by a factor of 10 over parsing. We believe this is also because the object creation protocol is slow. Usage of inline caches actually negatively impact performance because more objects are created during code generation.

Phase	Photon (ic)	Photon (noic)	Photon/V8
Parsing	9	11	48
Macro Expansion	0	0	2
Desugaring	0	0	2
Variable Analysis	0	0	2
Variable Scope Binding	0	0	4
Optimization	0	0	4
Code Generation	87	86	33

Table 8. Compilation Time Breakdown in % for controlflow-recursive.js

2.2.3 Discussion

Both execution time and compilation time performance are determined mostly by the limited time we had to optimize the implementation, not by limitations in the design. We provide here possible solutions to the problems identified and we show that with reasonable efforts the system could be made *pratically* fast.

The previous results indicate that object creation and string internalization are major bottlenecks of the current implementation. Object creation could be sped up using a caching strategy for the initial creation of the object and subsequently for the field initialization. String internalization can be made amortised constant time instead of linear time by using a hash map. Those two problems will be addressed in the near future since the implementation effort required is minimal.

Once all the common object model operations will be appropriately cached, the next step to reduce the execution performance gap between V8 and our system will be to provide optimized instructions for constant arithmetic operations and combine test in if expressions with the code generated for the conditional branching. This should also mitigate the absence of a register allocator by preventing access to the stack for binary operations with a constant.

Compilation time can be reduced by preencoding patterns of instructions for each corresponding AST node. Encoding is currently performed by making calls to the in-memory assembler for each instruction to encode. Encoding currently allocates many objects in the heap. Preencoded patterns could be exposed as functions that would patch values in a array mostly composed of integers. This will reduce self-hosted and v8-hosted compilation time since the number of created objects will be drastically reduced.

The compilation time breakdown for v8-hosted compilation shows that parsing is the dominant factor time-wise, followed closely by code generation. Further inquiry in the runtime behavior of OMeta grammars will be needed to see if this is something that could be optimized with changes to the grammar or with a more sophisticated OMeta compiler or if limitations in the design of OMeta would prevent such optimizations.

2.3 Experience Report

This experience taught us that bootstrapping a new self-hosted system is a compromise between simplicity of implementation and performance of the resulting system. Although our current implementation is too slow to serve for its own bootstrap, being able to bootstrap the system on V8 in 30 seconds allows fast experiments to be made to determine the origin of performance bottlenecks. Our previous experience with Tachyon was that 3 to 8 minutes were required for bootstrap, drastically reducing the speed at which experiments could be performed. In our opinion, this was mostly caused by the size of the code base and the really rich internal representation used to represent code, which required a huge number of objects to be allocated.

This suggests that when bootstrapping a self-hosted system, the first thing to do is to keep the implementation as simple as possible to minimize the quantity of code to compile. Bootstrapping time can be minimized by borrowing performance from existing optimized implementations, in our case, the V8 implementation for compiling the system and the C compiler to provide method behavior for objects. Should an existing optimized implementation did not exist, a fast compiler could be written in C. Compilation time can then be reduced by optimizing the bootstrap compiler. In our case, it is the same as the self-hosted compiler so this optimization benefits both. Once boostrap is near instantaneous, experiments can be performed to find and address performance bottlenecks until performance becomes a non-issue for all common development tasks.

For future work on Photon, the first thing to optimize would be the compilation speed on V8 by reducing the number of objects created, then addressing the speed bottlenecks introduced by OMeta until the bootstrap on V8 is near instantaneous. The next thing to optimize will be the self-hosted compilation speed until it is also near instaneous for incremental modification of the live system. That will allow fast experiment to be made until self-hosted bootstrap is near instantaneous. At that time, the system will be truly independent of existing implementations and free to evolve by itself to explore different implementation strategies. Boostrapping speed will not be a limiting factor in the exploration of VM implementation.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC), the Fonds Québécois de la Recherche sur la Nature et les Technologies (FQRNT) and Mozilla Corporation.

A. Appendix

A.1 JavaScript reserved properties

big table with all the reserved properties

References

[1] M. Chevalier-Boisvert, E. Lavoie, M. Feeley, and B. Dufour. Bootstrapping a self-hosted research virtual machine for javascript: an experience

report. In *Proceedings of the 7th symposium on Dynamic languages*, DLS '11, pages 61–72, New York, NY, USA, 2011. ACM.

```
function Map() {
    return this;
}

(function () {
    function indexOf(m, key) {
        var length = m[@length_offset]*2;
        for (var i = first_entry_offset; i < length; i += 2)
            if (m[@i] === key) return i;
        return -1;
    }
    function capacity(m) {
        return (m[@-3]/sizeof_ref - first_entry_offset)/entry_size;
    }
    function payload_size(capacity) {
        return (capacity*entry_size + first_entry_offset)*sizeof_ref;
    }
    function extend(m, cap) {
        var that   = m[@-5];
        var copy   = that.__clone__(payload_size(cap));
        var length = that[@-3] / sizeof_ref;

        for (var i = 0; i < length; ++i)
            copy[@i] = that[@i];

        m[@-5] = copy;
        return copy;
    }

    var length_offset      = 0;
    var first_entry_offset = length_offset + 1;
    var init_nb       = 10;
    var entry_size    = 2;
    var sizeof_ref    = this.__ref_size__();
    var init_payload = payload_size(init_nb);

    Map.prototype = Map.prototype.__clone__(
        (first_entry_offset + entry_size)*sizeof_ref
    );
    Map.prototype[@0] = 0;

    Map.prototype.__new__ = function () {
        var that  = this.__init__(0, init_payload);
        that[@-1] = this.__base_map__;
        that[@-2] = this;
        that[@length_offset] = 0;
        return that;
    }

    Map.prototype.__new_default__ = Map.prototype.__new__;
    Map.prototype.__base_map__ = Map.prototype[@-1].__new__();

    Map.prototype.get = function (k) {
        var that = this[@-5];
        var i = indexOf(that, k);
        if (i >= 0) return that[@i+1];
        else return undefined;
    };

    Map.prototype.has = function (k) {
        return indexOf(this[@-5], k) >= 0;
    };

    Map.prototype.set = function (k, v) {
        var that = this[@-5];
        var i = indexOf(that, k);

        if (i >= 0) return that[@i+1] = v;

        var length = that[@length_offset];
        var cap    = capacity(that);

        if (length === cap) that = extend(this, 2*cap);

        var i = 2*length + first_entry_offset;
        that[@i]     = k;
        that[@i + 1] = v;
        that[@length_offset]++;
        return v;
    };

    Map.prototype.delete = function (k) {
        var that = this[@-5];
        var i = indexOf(that, k);
        if (i < 0) return false;

        var length = that[@length_offset];
        var last = 2*(length-1)+first_entry_offset;

        if (i !== last) {
            that[@i]   = that[@last];
            that[@i+1] = that[@last + 1];
        }

        that[@length_offset]--;
        return true;
    };
})();
```

Figure 2. Extended JS Map implementation

```
var count, reset;

function instr(f) {
    var counter = 0;

    var g = function () {
        counter++;
        return f.call(this);
    };

    revert = function () {
        return f;
    };

    count = function () {
        return counter;
    };

    return g;
}

Object.prototype.__new__ = instr(
    Object.prototype.__new__
);
eval("1+2");
print("Object.prototype.__new__ called "
    + count() + " times");
Object.prototype.__new__ = revert();
o = {};
```

Figure 3. Dynamic profile of the number of regular objects created

```
var revert;

function instr_gen_send(f)
{
    var g = function (nb, rcv, msg, args, bind_helper)
    {
        print("Generating send '" + msg + "'");
        return f.call(this, nb, rcv, msg, args, bind_helper);
    }

    revert = function ()
    {
        return f;
    };

    return g;
}
PhotonCompiler.context.gen_send = instr_gen_send(
    PhotonCompiler.context.gen_send
);
eval("function fib(n) {" +
        "if (n<2) return n;" +
        "return fib(n-1)+fib(n-2); " +
     "}");
print(fib(10));
PhotonCompiler.context.gen_send = revert();
```

Figure 4. Static profile of message sends generated

```
(var fib (function (n)
    (if (< n 2)
        n
        (+ (fib (- n 1)) (fib (- n 2))))))
(print (fib 40))
```

Figure 5. Lisp-inspired syntax example for JS