Unpublished workshop paper for RESoLVE13 - March 16, 2013 - Houston, Texas

Harnessing Performance for Flexibility in Instrumenting a Virtual Machine for JavaScript through Metacircularity

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

Abstract

The limited reflexion features of the JavaScript (JS) language [5] on object operations and function calls has forced researchers, on tasks requiring run-time instrumentation, either to laboriously instrument production VMs or come up with *ad hoc* source-to-source translation schemes for each problem at hand. This paper shows that, by systematizing the second approach, it is possible to provide a dynamic run-time instrumentation of the object operations and function calls by running a metacircular VM on top of a performance-oriented JIT VM, without having to modify its source code. Our implementation, Photon, achieves a performance 2x slower than a state-of-the-art interpreter and 50x slower than a state-of-the art JIT implementation. The combination of simplicity of usage and efficiency of our system renders obsolete the necessity of instrumenting production VMs for instrumentation tasks that require interpreter-level performance and solves the core technical issues behind many source-to-source translation schemes.

1. Introduction

Run-time instrumentation of JavaScript is currently used for widely different purposes. Notable examples are automatic benchmark extraction from web applications [9], empirical data gathering about the dynamic behavior of Web applications [8] and access permission contracts enforcement [4]. All these examples require instrumentation of object operations, such as property accesses, updates and deletion. They also require instrumentation of all function calls made to global functions, object methods, or function references, either directly or indirectly through their `call` or `apply` method.

The standard semantics of JavaScript has no reflexion feature that completely covers those use cases. Object-method and global-function calls can be wrapped in closures to provide pre- and post-call instrumentation, providing a partial solution. However, direct calls to function references and object operations cannot be intercepted. To work around this limitation, two main approaches are used.

In the first approach, a production VM, usually written in C++, is instrumented to provide hooks on selected operations. It is increasingly complex on modern JS VMs because those VMs are optimized for performance. The resulting instrumented VM then becomes a burden to maintain up-to-date with its evolving counterpart.

In the second approach, an *ad hoc* source-to-source translator and run-time library are written for the problem at hand and the system is expected to run on top of a production VM. Depending on implementation choices, it can be more or less complicated to guarantee that instrumented objects cooperate well with the rest of the system. Moreover, the performance of the resulting system can be abysmal if care is not taken in implementing the instrumentation.

In this paper, we present Photon, a framework following the second approach that aims for usage simplicity, dynamism, and efficiency. It achieves simplicity by reifying object operations and function calls as methods on root objects. It allows their instrumentation at run time by replacing the function that implements their behavior. A single primitive operation, message-sending, is used as the focal point of implementation and optimization by mapping all reified operations to it. It allows dynamic instrumentation and specialization of run-time operations to information available at its call site and the current instrumentation state of the VM. Its specialized behavior is also reified as a method, called `__memoize__`, to provide an efficient way to implement uninstrumented operations and allow instrumentations to specialize their behavior at the call site of targeted operations. The optimization techniques presented in this paper allow Photon to achieve interpreter-level performance, around twice slower than the SpiderMonkey interpreter. We argue that it makes our approach efficient enough to replace instrumentations that were previously targeted at interpreters on production VMs.

Basing our design on the second approach allows us to take advantage of the availability of optimizations performed by the host VM. Our source-to-source translation is implemented by staging it before all `eval` calls, effectively providing a JIT-compiler. Moreover, we argue that viewing the second approach as implementing a metacircular VM provides both vocabulary to think about the organization of the system and inspiration from the existing literature for optimization techniques. We therefore refer to Photon as a metacircular VM for JavaScript.

This paper contains two original contributions: (1) an object representation exploiting the underlying VM's inline caches and dynamic object model to provide efficient virtualized operations and (2) the unification of the reified object operations and function calls around a single message-sending primitive that allows dynamic redefinition of their behavior and call-site specialization while preserving compatibility with the current version of JavaScript.

We present in turn, an overview of the components of the system, the object representation, the message-sending semantics, a compilation example and a preliminary performance evaluation.

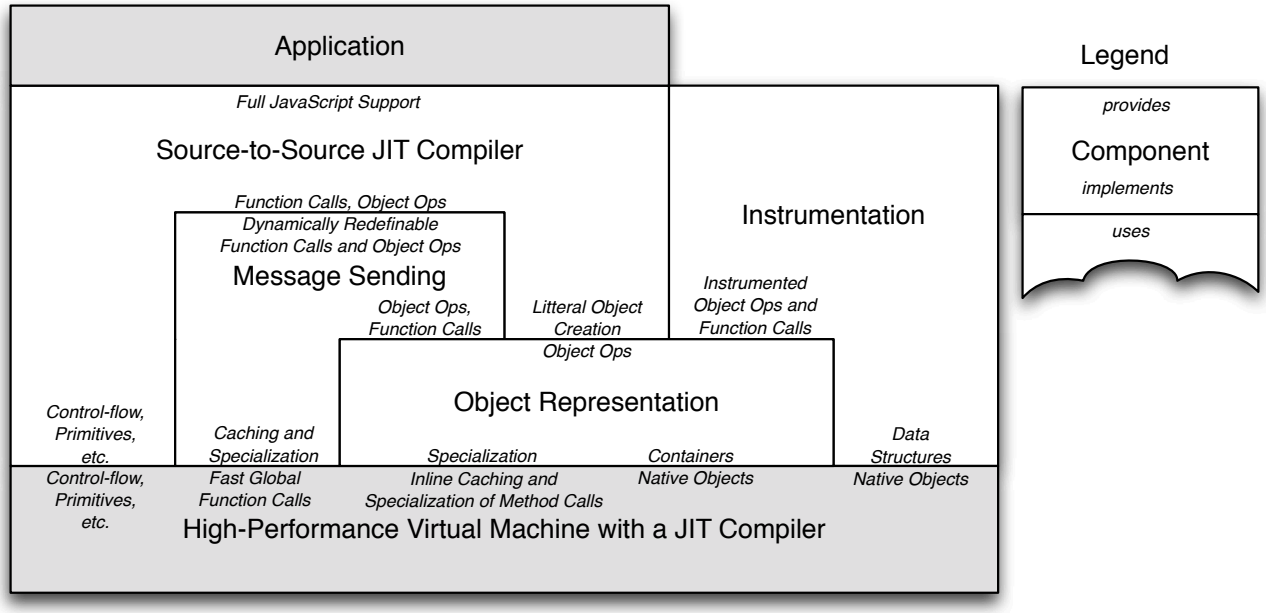

Figure 1. Components of Photon virtual machine and the features they provide, implement and use

2. Architecture

In a conventional JS setting, we can view an application as running over a host high-performance VM. The metacircular VM approach adds another layer between the two, allowing instrumentations to target the middle layer instead of the host VM. Figure 1 explicits the vertical interaction between layers by naming components and detailing which features are implemented using features of components below it as well as the feature they provide to the layer above. Note that the diagram represents a structural view of component interactions and not the execution model of the application. For example, after JIT compilation, the application code directly runs over the host VM, albeit in a different form, although it is shown as being over the source-to-source JIT compiler. Note that no meaning is associated to horizontal proximity.

We first present each component, in a bottom-up fashion, in terms of the abstractions they provide and the key features of their implementation. We then provide a simple example, in a top-down fashion, of a reified object operation and how each component implements its behavior.

2.1 Bottom-Up Overview

The object representation is the implementation of JS objects (including functions) from the point of view of the Photon VM. For efficiency, it uses native objects as property containers. The native objects are proxied with a second native object to allow encapsulating invariants of the implementation in proxy methods. It has the benefit of simplifying the implementation of instrumentations because it abstracts implementation details required for performance. It also allows object-specific instrumentation information to be stored on a proxy without risk of interference with the application properties. Object representation operations can be specialized to certain classes of objects for performance, such as indexed property accesses on arrays.

The message-sending layer builds on top of the object representation to provide dynamically redefinable object operations and function calls. This extra level of indirection allows specialization of those operations at their call site, depending of the call site data available, such as argument types and values, as well as the instrumentation state of the VM. It allows dynamic optimization of uninstrumented and instrumented operations. By presenting specialized operations as global functions, it allows the underlying VM to further specialize their behavior and even inline the operations in-place when possible. The invariants implied by the implementation of the message-sending layer are encapsulated in the object representation operations.

The source-to-source compiler translates the original JavaScript code to use the run-time environment of Photon. Non-reified elements, such as control-flow operations, primitive values and primitive operations are preserved. Object operations and function calls are translated to message sends and become dynamically redefinable. Literal object creation is translated to use the object representation. The source-to-source compiler is written in JavaScript and is therefore available at run-time. By staging it in front of every call to `eval`, it effectively provides a JIT compiler to Photon.

An instrumentation can redefine the behavior of object operations and function calls by replacing the corresponding method on a root object with an instrumented version using the object representation operations. The ability to completely replace a method provides maximum flexibility to instrumentation writers compared to being limited to a specific event before and after an operation. An instrumentation is written at the same level as the VM, which means that it has access directly to the execution environment of the VM and can use native objects as data structures.

2.2 Top-Down Example

To provide a more concrete example, suppose an application at some point performs an access to a property `p` on an `o` object the following way:

```
o.p;
```

Conceptually, this will be translated by the source-to-source compiler to an equivalent message send:

```
send(o, "__get__", "p");
```

In practice, for performance reasons, the call site for a message send has a unique name prefixed with `codeCache` followed by a unique numerical identifier. This identifier contains a global function whose body corresponds to the specialized operation. In this particular case, after its first call, it will be specialized to the `get` operations on a proxy object:

```
function codeCache0(rcv, msg, x0) {
    return rcv.get(x0);
}

codeCache0(o, "__get__", "p");
```

If the *__get__* method was subsequently redefined by an instrumentation at run-time, all message-send call sites for the *__get__* message would be reverted to the base case of a message send. Now if an instrumentation that would count all property accesses defined an instrumented version of the *__get__* method with a closure with a `__memoize__` function property, after the first call, the optimized version of the code cache would be:

```
var counter = 0; // Added by the instrumentation

function codeCache0(rcv, msg, x0) {
    counter++;
    return rcv.get(x0);
}

codeCache0(o, "__get__", "p");
```

The core idea here is that the message sending layer allows dynamic redefinition of optimized operations at every call site. In a hot section of the code, if the host VM then inlines the `codeCache0` global function and even the `get` method of the proxy object, most of the overhead of the dynamism will be eliminated.

The next sections explain in more details how the object representation and the message sending layer work.

3. Object Representation

Two insights led to the current design. First, on a well optimized VM, the most efficient implementation of a given operation in a metacircular implementation is frequently the exact same operation performed by the host VM. Second, method calls on JavaScript VMs are usually really fast. Therefore, we should provide operations as method calls on objects whose internal representation is as close as possible to the host object being implemented. It can be achieved by structuring the object representation as proxies to native objects with a particular constraint, which we believe has never been exploited before: *the prototype chain of the proxies should mirror the prototype chain of the native objects*.

The core idea is to associate a proxy object to every object in the system. In itself, this idea is not new. It has been advocated by Bracha and Ungar almost 10 years ago to provide meta-level facilities [3]. Proxy objects are also bound to appear in the next version of JavaScript and are currently supported by some major JS VMs. However to the best of our knowledge, it has never been suggested that the prototype chain of proxy objects should follow the prototype chain of proxied objects. As we will see, it opens the possibility of specializing and optimizing the object representation operations at runtime by attaching specialized methods at the appropriate places on the proxy prototype chain.

3.1 Basic Representation

The first and simplest object type in JavaScript is the object. It has properties and a prototype. A proxy object has a reference to the native object to intercept every operation that goes to the object. The prototype chain of proxies mirrors the object prototype chain. A JavaScript implementation, with `Object.prototype` as the root of all objects, is illustrated in Figure 2.

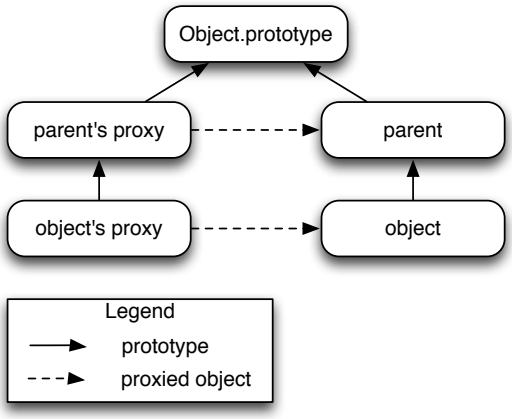

Figure 2. Basic object representation

An advantage of a representation like this one is that property accesses can be implemented directly as a native property access to the proxied object. It allows the host VM to do lookup caching. It even works for reified root objects, in this example, by considering `Object.prototype` the parent of all objects of the metacircular VM.

However, it does not work well with native types that can be created literally such as arrays, functions and regular expressions. These would require their prototype to be changed to another object at the creation site, ruining structural invariants assumed by the host VM. For those objects, the original native prototype is maintained and in cases where a lookup needs to be performed, it is done explicitly through the proxy prototype chain. This is illustrated with arrays in Figure 3.

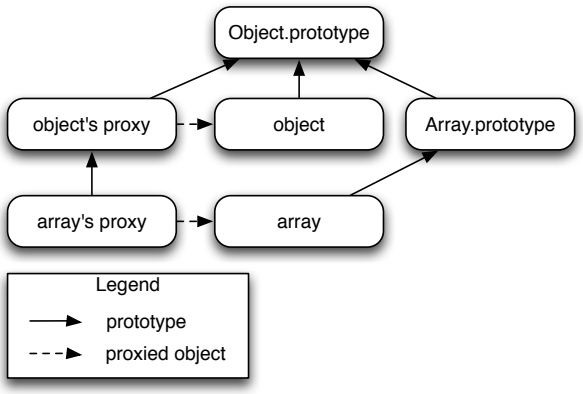

Figure 3. Special object representation

Given this structural representation for basic types, we can now define all object operations as methods on proxy objects as explained in Table 1. Given the current JavaScript *de facto* standard of accessing the prototype object with the `__proto__` name, if proper care is not taken in the property access method, the proxied

object will be returned instead of the expected proxy object of the parent. [1]

Table 1. Object representation operations examples and their interface

Operation	Interface	Example
Property access	`get(name)`	`o.get("p")`
Property assignation	`set(name, value)`	`o.set("p",42)`
Property deletion	**`delete`**`(name)`	`o.`**`delete`**`("p")`
Object creation	`create()`	`parent.create()`
Call	`call(rcv, ..args)`	`fun.call(global)`
Prototype access	`getPrototype()`	`o.getPrototype()`

Although proxies mirror native objects in their prototype chain, they do not mirror their properties. In fact, their properties can be fixed for the whole execution if one assumes that the semantics of object operations does not change. These facts allow proxies to adapt to dynamic circumstances by adding specialized methods at run time, which can be used for performance gains. We will demonstrate how to exploit this fact, first to specialize operations to a fixed number of arguments and then to a constant argument type or value. Those two specializations are independent and can be combined for further performance gains. To avoid name clashing and ease reading we adopt the convention that specialized methods share the same prefix as basic methods with the type or value information and or number of arguments following in the name.

3.2 Specialization on a Fixed Number of Arguments

The object representation design does not require a special calling convention for functions. However, for maximum performance gains in JavaScript, we would like to avoid using the `call` and `apply` native methods. We can do it by globally rewriting every function to explicitly pass the receiver object. This way, a specialized call operation can simply pass the references to the native function. An example implementation for a `call` operation specialized for one argument in addition to its receiver could be:

```
var fun = FunctionProxy(function ($this, x) {
    return x;
});

fun.call1 = function ($this, arg0) {
    return this.proxiedObject($this, arg0);
};

// Also add a call1 method on all proxies
// that might be called as a method!
```

Specializing one proxy operation requires us to specialize *all* objects for that particular operation to ensure that whatever receiver, function or object is called at a given site, a proper operation is supplied. Fortunately, the object-oriented nature of our chosen representation makes it easy. Only root proxies need to have an additional method and all other proxies will then implement the specialized operation.

3.3 Type or Value Specialization

In the same way, we can specialize an operation for a known value. For example, suppose we would like an optimized operation that accesses a stable property name. A proper specialized method could be written:

[1] For this particular reason, we would advocate for implementations to expose the prototype of an object through a method call instead of a property. We could still preserve backward compatibility for literal object definitions but once the object is created, the prototype should not be accessible or modifiable through the `__proto__` property.

```
var obj = root.create();

obj.set("foo", 42);

obj.get_foo = function () {
    return this.proxiedObject.foo;
};

root.get_foo = function () {
    return this.get("foo");
};
```

Notice that a method that falls back to the general case is provided to the root object to add support to all other proxies. The same idiom can be used for type specialization instead of value.

4. Message-Sending Semantics

The message-sending semantics builds on top of the object representation to allow the behavior of object operations and function calls to be dynamically redefined at their call site, which happens if an instrumentation redefines the behavior of the corresponding method. This two-level separation allows encapsulation of invariants of the implementation inside the object representation methods to simplify the instrumentation code. It also allows an efficient implementation of reified operations by allowing specialization of their behavior to their call site.

The semantics of message sends is progressively introduced with JavaScript code. First, its root in the method call semantics is shown because it is sufficient to reify object operations. Then by enriching it to add a causal connection between the function `call`'s method and all call site's behavior it becomes powerful enough to reify function calls. Finally, an efficient implementation using inline caches is presented and the reification of its specialized behavior at a call site through a `__memoize__` method is discussed. This reification allows instrumentations to specialize their behavior at the call site with the same mecanism used for specializing uninstrumented standard operations.

Unless specified, the pseudo-code follows the standard JavaScript semantics. We take advantage of the expressivity of the rest and spread operators (e.g. `..args`), bound to appear in the next revision. For simplicity, primitive values, missing values and error handling are omitted.

4.1 Reifying Object Operations

In its essence, a message send corresponds to the standard semantics of a method call in JavaScript, which is a property access through the prototype chain followed by a call:

```
function send(rcv, msg, ..args) {
    var m = rcv.get(msg);
    return m.call(rcv, ..args);
}
```

Given this semantics, the basic object operations of the language can be reified as methods. Opened object operations, JavaScript examples and their equivalent message sends are given in Table 2.

However, from the point of view of the user program, the `call` operation is opaque and cannot be modified. The semantics of JavaScript does not provide a means to intercept all calls. However, we can provide the capability of *dynamically* instrumenting function calls by augmenting the semantics of the send operation.

4.2 Reifying the Function Calls

In JavaScript, the `call` method on every function reifies the calling protocol and allows a program to call into a function at run time, as if it was done through the direct mechanisms. The exact behavior of a function call should be the same, whether it is called directly

Table 2. Object model operations and examples of their equivalent message sends

Object Model Operation	Example	Equivalent Message Send Example
Property access	`o.p`	`send(o,"__get__","p")`
Property assignation	`o.p=42`	`send(o,"__set__","p",42)`
Property deletion	`delete o.p`	`send(o,"__delete__", "p")`
Object litteral creation	`{p:42}`	`send({p:42}, "__new__")`
Creation with constructor	`new Fun()`	`send(Fun, "__ctor__")`

or indirectly. However, there is no causal connection between the state of the `call` method and the behavior of function calls. In other words, redefining the `call` method on `Function.prototype` does not affect the behavior of call sites.

We can establish this causal relationship with a slight modification of the send operation:

```
function send(rcv, msg, ..args) {
    var m = rcv.get(msg);
    var callFn = m.get("call");
    return callFn.call(m, rcv, ..args);
}
```

This semantics allows all function calls to be instrumented, simply by redefining the root function's call method. This way, all function calls can be intercepted, as long as they are mapped to message sends. An explanation of each compile-time occurrence of function calls as well as their equivalent message sends are given in Table 3.

Table 3. Call types and their equivalent message sends

Call Type	Explanation	Equivalent Message Send
Global	Calling a function in the global object. Ex: `foo()`	Sending a message to the global object. Ex: `send(global,"foo")`
Local	Calling a function in a local variable. Ex: `fn()`	Sending the `call` message to the function. Ex: `send(fn,"call")`
Method	Calling an object method. Ex: `obj.foo()`	Sending a message to the object. Ex: `send(obj,"foo")`
`apply` or `call`	Calling the `call` or `apply` function method. Ex: `fn.call()`	Sending the `call` or `apply` message. Ex: `send(fn,"call")`

4.3 Efficient Implementation

The core insight behind our implementation comes from seeing global function calls both as an optimized calling mechanism and as a dynamically specializable operation. They provide the same ability as code patching in assembly. On the current version of V8, when the number of expected arguments matches the number of supplied arguments, inlining the function at its call site becomes possible. If the global function is redefined at a later time, the call site will be deoptimized transparently. It is a really powerful mechanism because much of the complexity of run-time specialization is performed by the underlying host. We can simply piggyback on those optimizations.

For example, given the aforementioned semantics of message sending, sending the message `msg` to an object `obj` inside a `foo` function can be written this way:

```
function foo(obj) {
    send(obj, "msg"); // Equivalent to obj.msg();
}
```

The `send` function is a global function. We can replace it with another global function that is guaranteed to be unique, so it can be identified with the call site. In addition to the message to be sent, we can pass it a unique identifier that will be used to find the corresponding global function name, for later specialization of the call site:

```
function initState(rcv, dataCache, ..args) {
    ...Update ("codeCache" + dataCache[0])
    return send(rcv, dataCache[1], ..args);
}

var codeCache0 = initState;
var dataCache0 = [0, "msg"];

function foo(obj) {
    codeCache0(obj, dataCache0);
}
```

The `initState` function follows the same calling convention as the `send` function. Furthermore, `dataCache0` is an array, which means that the different states of the cache can use the array to store additional information.

After an initial execution, the second time around, the cache will hold an optimized version of the operation. Given this new definition, the cache state might be equivalent to:

```
var codeCache0 = function (rcv, dataCache) {
    return rcv.get("msg").call(rcv);
};
var dataCache0 = [0, "msg"];

function foo(obj) {
    codeCache0(obj, dataCache0);
}
```

Apart from the indirection of the global function call, this example is optimal with regard to the object representation definition we have. If the underlying runtime chooses to inline the global function, the cost of the indirection will be effectively eliminated.

4.3.1 Memoized Methods

Memoization is usually associated with functional programming and entails trading space-efficiency for time-efficiency by remembering past return values of functions with no side-effect. By analogy, we will say that a memoized method is a method that performs the same operation, albeit possibly more efficiently by exploiting run-time information (e.g., types or argument count). This particular functionality became necessary when trying to efficiently implement the JavaScript object operations in our system because they are reified as methods.

The basic idea is to allow a method to inspect its arguments and receiver to specialize itself for subsequent calls. The first call is always performed by calling the original function while all subsequent calls will be made to the memoized function. A function call defines its memoization behavior by having a `__memoize__` method.

There is an unfortunate interaction between memoization and the reification of the call protocol. A further refinement specifies that memoization can only occur if the call method of the function has not been redefined. Otherwise, the identity of the function

passed to the call method would not be the same. To preserve identity while allowing memoization, the behavior of the cache can be different depending on the state of the Function.prototype's `call` method. If its value is the default one, the identity of the function is not important and memoization can be performed. Otherwise, memoization will be ignored. This definition has the advantage that one can temporarily redefine the calling method without penalty after the original method has been restored.

4.3.2 Cache States and Transitions

The last missing piece is the precise definition of the behaviors of the inline caches and the conditions that trigger those behaviors. We use a state-machine formalism to present the different behaviors associated with inline caches and the triggers that are responsible for the transitions between those behaviors. In our formalism, due to the nature of synchronous message sends, a state transition occurs before the event has been fully processed. However the processing of the event is not influenced by it.

To simplify invariants tracking, we decided to always perform lookups for method calls, i.e., method calls are always a `get` followed by a `call`. This is a reasonable choice if the object representation can piggyback on the host optimizations. The other important operation was to allow specialization of object operations, through memoized methods. There are therefore two states in addition to the initial state of the cache, as explained in Table 4.

Table 4. Cache states

Cache states	Explanation
Initial State	Perform the full send operation.
Regular method call	Look up method, then call.
Memoized method call	Method-specific behavior.

Transitions between states happen on message-send and object-operation events. An insight was to realize that we could under-approximate the tracking of invariants and conservatively invalidate more caches than minimally required. As long as the operations triggering the invalidation of caches are infrequent, the performance impact should be minimal. We therefore track method values cached in memoized states by name without consideration for the receiver object. If a method with the same name is updated on any object, all caches with a given message name will be invalidated. Also, if the `call` method on the `Function.prototype` object or any method with the `__memoize__` name is updated, *all* caches will be invalidated. This way, we only need to track caches associated with names. The upper bound on memory usage for tracking information is proportional to the number of cache sites.

There is no state associated with a redefined `call` method. In that particular case, all caches will stay in the initial state and a full message send will be performed. Figure 4 summarizes those elements in a state diagram. A more detailed explanation of every event and transition conditions is given in Table 5 and Table 6.

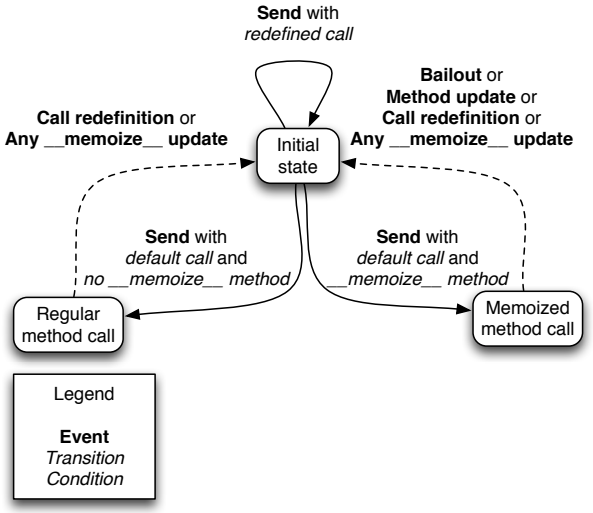

Figure 4. Cache States and Transitions

Table 5. Cache Events

Cache Events	Explanation
Send	A message is sent to a receiver object.
Call redefinition	The `call` method on `Function.prototype` is redefined.
Any memoized redefinition	Any `__memoize__` method is being redefined.
Bailout	A run-time invariant has been violated.
Method redefinition	An object with a method with the same name has its method being updated.

Table 6. Cache Transition Conditions

Cache Transition Condition	Explanation
Default call	`Function.prototype call` method is the same as the initial one.
Redefined call	`Function.prototype call` method is different than the initial one.
No `__memoize__` method	No method named `__memoize__` has been found on the method to be called.
`__memoize__` method	A method named `__memoize__` has been found on the method to be called.

5. Compilation Example

To illustrate how those different elements work together in practice, consider following example:

```
(function () {
    var o = {};
    o.foo = function () { return this.bar; };
    o.bar = 42;

    for (var i = 0; i < 2; ++i) {
        o.foo();
    }
})();
```

In this example, an anonymous function is called right after being created to provide a lexical scope, which means that the `o` and `i` variables are local to the function. In this scope, we create an `o` empty object, which has the root object of the metacircular VM for prototype. Then this object is extended with a `foo` method. This method returns the `bar` property of the receiver object. We then create and initialize the `bar` property of the `o` object. Finally, we call the `foo` method two times to give it the chance to specialize the call site, both of the `foo` call and the `bar` property access inside the `foo` method.

In addition to what has been discussed previously, additional details appear in the compilation result:

- *Type information in data caches*: During compilation, known types which directly correspond to abstract syntax tree nodes are preserved. It allows the runtime to exploit stable information. For example, in `dataCache1`, the `"string"` type allows the runtime to know the property access is to a constant string name.
- *Root objects are different from the host root objects*: `root.function`, `root.object` and `root.global` virtualize the object model's root objects to avoid interference with the host objects.
- *Functions have an extra `$closure` parameter*: This extra parameter is used to pass the proxy to the function to the code or `dataCache` information for the implementation to send the cache state to the cache function behavior.

The compiled code has been weaved with the original code in comments for clarity:

```
(codeCache0 = initState);
(dataCache0 = [0, "__new__",[]]);
(codeCache1 = initState);
(dataCache1 = [1,"__get__",
    ["this","string"]]);
(codeCache2 = initState);
(dataCache2 = [2,"__new__",[]]);
(codeCache3 = initState);
(dataCache3 = [3,"__set__",
    ["get","string","icSend"]]);
(codeCache4 = initState);
(dataCache4 = [4,"__set__",
    ["get","string","number"]]);
(codeCache5 = initState);
(dataCache5 = [5,"foo",["get"]]);
(codeCache6 = initState);
(dataCache6 = [6,"__new__",[]]);
(codeCache7 = initState);
(dataCache7 = [7,"call",[]]);

// (function () {
(codeCache7((codeCache6(
    root.function, dataCache6,
    (new FunctionProxy(function ($this,$closure) {
        var o = undefined;
        var i = undefined;
        // var o = {};
        (o = (codeCache0(root.object, dataCache0,
                (root.object.create()))
        );
        // o.foo = ...
        (codeCache3(o, dataCache3, "foo",
        //     ... function () { return this.bar; };
            (codeCache2(
                root.function, dataCache2,
                (new FunctionProxy(function (
                    $this,$closure) {
                    return (codeCache1(
                        $this, dataCache1, "bar")
                    );
                }))))));
        // o.bar = 42;
        (codeCache4(o, dataCache4, "bar", 42));
        // for ...
        for ((i = 0); (i < 2); (i = (i + 1))) {
            // o.foo();
            (codeCache5(o, dataCache5));
        }
    }))
// })();
)), dataCache7, root.global));
```

After execution, the inline caches at `codeCache1` and `codeCache5` will be respectively in a memoized state and a method call state, which correspond to the following behaviors:

```
codeCache1 = function ($this,dataCache,name) {
    return $this.get(name);
}

codeCache5 = function($this,dataCache) {
    return $this.get("foo").call0($this);
}
```

In the last case, we can see that the `call` method has been specialized for no arguments, exploiting optimization opportunities offered by our object representation. This example therefore summarizes the unification of object model operations to message sends, their efficient implementation and a novel object representation that can dynamically adapt itself to information available at runtime.

6. Performance

Our current results show that the baseline performance (without instrumentation) is around twice as slow as the SpiderMonkey interpreter on both the V8 and SunSpider benchmarks. The baseline memory usage in most cases, is also around twice that of V8. A basic instrumentation slows the system by an additional factor of three but has almost no effect on memory usage, while the same instrumentation optimized for speed can have negligible impact both on run-time performance and memory usage compared to the baseline.

6.1 Methodology

We compare three different systems:

- Photon running over V8 (V8's full optimizations)
- V8 (full optimizations)
- SpiderMonkey interpreter (JIT disabled)

V8 was chosen to host Photon because preliminary tests showed the system to be faster on it. The additional speed is attributed to the ability of the runtime to perform function inlining and on-stack-replacement, as well as to the presence of fast garbage collectors. Other VMs are catching up on features and the current focus seems to be on method-based Just-In-Time compilers so we anticipate that in the near future they could probably be used interchangeably. We compare Photon to V8 to quantify the performance cost incurred by our approach. We finally compare to a popular state-of-the-art interpreter, SpiderMonkey, to argue that our approach can be used wherever a manual instrumentation of that interpreter could have been performed.

To assess performance, we use the V8 benchmark suites, since it is one of the *de facto* standards to compare JS VM performance. We used the original methodology of the benchmark suite to make our results comparable to other published results for other systems. All benchmarks were run unmodified.

We focus on two metrics, running time and the maximum heap size to respectively measure run-time performance and memory usage. Running time is measured using the score given by the benchmark suite. Higher scores mean a faster system (less execution time). Memory usage is used to estimate the overhead of Photon compared to V8, but no attempt was made to evaluate the memory usage of the SunSpider interpreter.

We present results in two different groups, the baseline performance and the instrumented performance. The baseline performance is used to measure the minimal overhead of the approach since it determines its viability, regardless of other characteristics. The instrumented performance is used to measure the impact of

instrumentation on the baseline performance. It is common knowledge that instrumenting an interpreter has little impact over its overall performance (this was verified by Richards et al. when they instrumented JavaScriptCore [8]). However, our approach is more sensitive to instrumentation. We therefore quantify its impact.

The chosen instrumentation counts the number of occurrence at run time of property accesses, assignments and deletions. The original operation is inlined in the replacement method instead of being called. We chose this particular instrumentation because it is simple, it covers important object model operations and it was actually used to gather information about JS (it can be used to reproduce the object read, write or delete proportion figure from [8]).

All results were obtained on a MacBook Pro running OS X version 10.7.5 with a 2.2 GHz Intel Core i7 processor and 8 GB of 1333 MHz DDR3 Ram. We used V8 revision 12808 and SpiderMonkey version 1.8.5+ 2011-04-16. The results are intended to give an order of performance for the approach. Therefore, it was not deemed necessary to provide confidence intervals and means of multiple runs, given that the test harness already run the benchmarks multiple times. Although some variations between runs were noticed, they were sufficiently stable to not affect our arguments.

For conciseness, abbreviations are used in tables. They are listed in order of appearance:

- Pn: Photon
- SM: SpiderMonkey
- V8: V8
- Pn-spl: Photon with our simple instrumentation
- Pn-fast: Photon with an equivalent instrumentation optimized for speed

6.2 Baseline Performance

6.2.1 Running Times

Table 7 shows the baseline performance for V8 benchmarks. The results indicate that Photon obtains an overall score within a factor of 2 of the SpiderMonkey interpreter. On three out of the eight benchmarks Photon is faster and in two cases by almost of factor of two. In other cases, the SpiderMonkey interpreter is between 2 and 3.5 times faster, except for the Splay benchmark where it is 13 times faster.

This last case seems to be a pathological case for the basic optimizations performed on property access, assignation and update. As shown later, for this particular benchmark, the instrumented version of Photon is three times faster than the non-instrumented version. This can be explained by the fact that the instrumented version *removes* the optimizations attempted. However, removing the same optimizations for all benchmarks gives an overall score 30% inferior with some benchmarks almost four times slower, except for Splay.

Table 7. Baseline performance on V8 benchmarks

Benchmark	Pn	SM	V8	V8/Pn	SM/Pn
Crypto	529.0	348.0	17025.0	32.2	0.7
DeltaBlue	82.8	249.0	19306.0	233.2	3.0
EarleyBoyer	738.0	808.0	34170.0	46.3	1.1
NavierStokes	908.0	564.0	20947.0	23.1	0.6
RayTrace	156.0	560.0	19442.0	124.6	3.6
RegExp	441.0	781.0	3902.0	8.8	1.8
Richards	120.0	219.0	14149.0	117.9	1.8
Splay	118.0	1508.0	5850.0	49.6	12.8
V8 Score	270.0	524.0	14002.0	51.9	1.9

6.2.2 Memory Usage

The heap size is measured from garbage collection traces. Memory usage in Table 8 is acceptable, given that every run-time object has an associated proxy with most cases using less than a factor of two in memory and a worst case of around 6.5. The high memory usage in the EarlyBoyer case can be explained by the memory overhead of every inline cache site having an associated array.

Table 8. Baseline memory usage (in MB) on V8 benchmarks

Benchmark	Pn	V8	Pn/V8
Crypto	56.0	20.0	2.8
DeltaBlue	33.0	20.0	1.6
EarleyBoyer	128.0	20.0	6.4
NavierStokes	29.0	19.0	1.5
RayTrace	35.0	20.0	1.8
RegExp	54.0	22.0	2.5
Richards	28.0	18.0	1.6
Splay	84.0	97.0	0.9

6.3 Instrumented Performance

Two versions of the object operations instrumentation are analyzed. The simple version does not exploit the memoization protocol and corresponds to the straight-forward implementation: incrementing a counter and calling the corresponding object representation method. The fast version inlines the instrumentation operation inside the optimized version of object operations for speed.

The first version is intended to measure the performance that can be expected from a quickly developed instrumentation while the second one is intended to measure the performance impact of the instrumentation operations alone. This is therefore a low-barrier high-ceiling example and illustrates the flexibility that can be gained when the choice of aiming for performance is left to users of the system.

Table 9 shows the impact of instrumentation on the baseline performance. The fast version has negligible impact with some results slightly faster, mostly due to natural variation of results. However, the simple version can be as much as 18 times slower on some benchmarks.

Table 9. Instrumented performance on V8 benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn/Pn-spl	Pn/Pn-fast
Crypto	529.0	41.4	566.0	12.8	0.9
DeltaBlue	82.8	36.2	103.0	2.3	0.8
EarleyBoyer	738.0	162.0	767.0	4.6	1.0
NavierStokes	908.0	51.4	871.0	17.7	1.0
RayTrace	156.0	85.1	158.0	1.8	1.0
RegExp	441.0	324.0	476.0	1.4	0.9
Richards	120.0	30.5	113.0	3.9	1.1
Splay	118.0	453.0	117.0	0.3	1.0
V8 Score	270.0	91.2	281.0	3.0	1.0

Table 10 shows the memory overhead, compared to the baseline memory overhead. For most benchmarks, this is negligible. However for the Splay benchmark a ratio of 1.7 is observed, that is the simple instrumentation uses 1.7 times the memory of the baseline version. Although we have not uncovered the specific cause yet, we speculate that it comes from V8 duplicating methods to facilitate specialization.

6.4 Interpretation

Our current optimizations lack stability across benchmark results. A better predictor of whether to optimize or not would give more even results, especially for the Splay benchmark. We believe this

Table 10. Instrumented memory usage (in MB) on V8 benchmarks

Benchmark	Pn	Pn-spl	Pn-fast	Pn-spl/Pn	Pn-fast/Pn
Crypto	56.0	55.0	56.0	1.0	1.0
DeltaBlue	33.0	32.0	32.0	1.0	1.0
EarleyBoyer	128.0	103.0	128.0	0.8	1.0
NavierStokes	29.0	29.0	32.0	1.0	1.1
RayTrace	35.0	34.0	35.0	1.0	1.0
RegExp	54.0	54.0	54.0	1.0	1.0
Richards	28.0	28.0	28.0	1.0	1.0
Splay	84.0	139.0	88.0	1.7	1.0

can be done within the framework presented. Furthermore, the memory overhead seems acceptable for instrumentation tasks given our multi-gigabytes of RAM in today's machines. Instrumentation can have a negligible impact both on performance and memory usage, as long as the instrumented operations are optimized enough. Otherwise, the choice of trading performance of a faster development time can still be made with an impact on performance that can be reasonable in most cases.

7. Related Work

Although there are existing systems for JavaScript targeting JavaScript as their runtime, such as Google Caja to enforce security invariants [1], Google Traceur to support the next version of JavaScript on existing VMs [2] and JSBench to record execution traces for automatic benchmark generation [9], we believe our combination of simplicity, flexibility and efficiency as well as the focus on instrumentation is unique.

The initial inspiration for optimizing sends with global functions that change during execution was drawn from the lazy function definition pattern as explained by Peter Michaux [6]. In this pattern, after the initial setup performed by a function, a new function without the initialization code can replace the original function to provide an efficient operation. We believe this is the first time this pattern is used as an inline cache in the literature.

The particular choice of message-sending as a foundation was motivated by its successful application in Smalltalk and Self to achieve a dynamic, open and fast implementation. The initial work on unifying an object model around a message-sending primitive came from the Open, Extensible Object Models from Piumarta and Warth [7].

In choosing the systems to compare, we eliminated a few current alternatives. We assume that when faced with the task of instrumenting production code to obtain run-time data, manually instrumenting a JIT-compiler would be deemed too complex to be cost-effective in terms of development time. At the time of writing, JavaScriptCore's low-level interpreter became available and replaced the original interpreter that was instrumented by Richards et al. [8] in WebKit. We argue that the only real instrumentation alternative right now would be SpiderMonkey's interpreter because the JavaScriptCore low-level interpreter is implemented in an assembly dialect to obtain performance gains.[2] As this new interpreter matures, we speculate that its complexity will increase, negating most of the simplicity usually attributed to interpreters. Finally, we do not show performance results for Narcissus, Mozilla's JavaScript in JavaScript interpreter, because the latest version would not run either the SunSpider or V8 benchmarks.

[2] In our tests on the V8 benchmarks, the JavaScriptCore low-level interpreter was roughly three times faster than SpiderMonkey's interpreter. How much of those speed gains would remain in presence of instrumentation is unknown.

8. Limitations

Accessing the `__proto__` property leaks the internal representation. This can be solved at a substantial performance cost by testing every property access. Alternatively, it can be mitigated with no run-time penalty by detecting, at compile-time, accesses to the `__proto__` property and calling the object representation `getPrototype` method instead. However, the possibility of dynamically generating the `__proto__` name render it unsound. It is yet to be seen if this actually happens in practice.

Meta-methods can conflict with application methods if they have the same name. This limitation will be solved in the next version of the standard, when unforgeable names will be available in user space. Until then, we can rely on rarely used names to minimize possible conflicts with existing code.

Setting the `__proto__` property throws an exception. This might be fixed by invalidating all caches should the prototype of an object change. A more sophisticated mechanism could be devised if the operation is frequent.

Operations on *`null`* or *`undefined`* might throw a different exception because they might be used as base objects for an object representation method. The exception will say that the object representation is missing instead of the property. This problem only happens for incorrect programs because otherwise an exception would still interrupt it. We choose not to handle this case.

Functions passed to the standard library are wrapped to remove the extra arguments introduced by our compilation strategy. However, when called by the standard library methods, a direct call is made. Should the need to intercept those calls arise, the wrappers could perform a message send instead of a direct call.

A major limitation of this approach is that all dynamically evaluated code needs to be intercepted in order to be translated and maintain invariants of the system. It is non-trivial in a browser setting.

9. Conclusion and Future Work

We demonstrated how object operations and function calls could be reified by basing them on a single message-sending primitive, with a performance similar to a state-of-the-art interpreter. Novel and efficient object representation and message-send implementations were devised. These were shown to be key to a metacircular approach to instrumenting JS VMs.

The design of the current system has focused on flexibility; some additional performance gains could be obtained with known optimizations. For example, no property-access specialization is currently performed.

The cost of various aspects of the design should also be determined. In particular, the cost of the object representation could be isolated by compiling directly to it, thus avoiding the indirection of message sends. All available compile-time information could be translated to proper specialized object representation methods. The performance cost of the reification and the dynamicity could then be separated.

Finally, support for the browser Document Object Model and a proper integration with a web browser will be done to test the system on real web applications.

Acknowledgments

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

Thanks to all the anonymous reviewers for the invaluable feedback on earlier versions of this paper.

References

[1] Google caja. http://code.google.com/p/google-caja/, December 2012.

[2] Google traceur. http://code.google.com/p/traceur-compiler/, December 2012.

[3] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 2004 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 331–344, New York, NY, USA, 2004. ACM. ISBN 1-58113-831-8. doi: http://doi.acm.org/10.1145/1028976.1029004. URL http://doi.acm.org/10.1145/1028976.1029004.

[4] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 111–122, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103671. URL http://doi.acm.org/10.1145/2103656.2103671.

[5] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, Dec. 1999.

[6] P. Michaux. Lazy function definition pattern. http://michaux.ca/articles/lazy-function-definition-pattern, December 2012.

[7] I. Piumarta and A. Warth. Self-sustaining systems. chapter Open, Extensible Object Models, pages 1–30. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-89274-8. doi: 10.1007/978-3-540-89275-5_1. URL http://dx.doi.org/10.1007/978-3-540-89275-5_1.

[8] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12. ACM, 2010.

[9] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of javascript benchmarks. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 677–694, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048119. URL http://doi.acm.org/10.1145/2048066.2048119.