Portable and Efficient Run-time Monitoring of JavaScript Applications using Virtual Machine Layering

Erick Lavoie[1], Bruno Dufour[2], and Marc Feeley[2]

[1] McGill University, Montreal, Canada
erick.lavoie@mail.mcgill.ca *
[2] Université de Montréal, Montreal, Canada
{dufour,feeley}@iro.umontreal.ca

Abstract. Run-time monitoring of JavaScript applications is typically achieved either by instrumenting a browser's virtual machine, usually degrading performance to the level of a simple interpreter, or through complex *ad hoc* source-to-source transformations. This paper reports on an experiment in layering a portable JS VM on the host VM to expose implementation-level operations that can then be redefined at run-time to monitor an application execution. Our prototype, Photon, exposes object operations and function calls through a meta-object protocol. In order to limit the performance overhead, a dynamic translation of the client program selectively modifies source elements and run-time feedback optimizes monitoring operations. Photon introduces a 4.7× to 191× slowdown when executing benchmarks on popular web browsers. Compared to the Firefox interpreter, it is between 5.5× slower and 7× faster, showing the layering approach is competitive with the instrumentation of a browser VM while being faster and simpler than other source-to-source transformations.

Keywords: JavaScript, Virtual Machine, Runtime Monitoring, Performance Evaluation, Optimization, Metaobject Protocol

1 Introduction

JavaScript (JS), the *de facto* language of the web, has recently gained much popularity among researchers and practitioners alike. In particular, due to the highly dynamic nature of the language, there is a growing interest in observing the behavior of JS programs. For instance, run-time monitoring is being used for widely different purposes, such as gathering empirical data regarding the dynamic behavior of web applications [10], automatically extracting benchmarks from web applications [11], and enforcing access permission contracts [6].

Common profiling tasks in JS, such as intercepting all object operations or function calls, are difficult to achieve in a portable and efficient manner. A popular approach consists of modifying a production virtual machine (VM). While

* This work was done at Université de Montréal.

this approach guarantees a high level of compliance with the source language, it suffers from some important drawbacks. Most modern JS implementations are production-quality VMs that are optimized for performance and thus difficult to modify. Generally, this approach also binds the profiling system to a single VM, and therefore greatly limits the portability of the approach. Moreover, modifications to the VM codebase must evolve as the VM is being developed upstream, which can happen at a rapid pace. As a result, many attempts to modify a JS VM are punctual efforts that are abandoned shortly thereafter [3,8,10].

The most popular alternative approach for instrumenting JS programs consists of implementing an *ad hoc* source-to-source translator and runtime library tailored to the problem at hand. While this approach is easier to maintain and more portable than instrumenting a VM, implementing a correct source-to-source transformation is deceptively difficult in practice, even for seemingly simple tasks. For instance, instrumenting all object creations also requires instrumenting all function calls because any function call could potentially be a call to `Object.create` through an alias. Other dynamic constructs in JS, such as `eval`, are notoriously difficult to instrument while guaranteeing that the observed behavior of the program will remain unaffected. Also, JS programs can easily redefine core operations from `Object` and `Array`. Such modifications are difficult to handle. A profiler that is unaware of such redefinitions could behave incorrectly, or worse, cause a change in the observed behavior of the profiled program. Finally, the profiler code itself must maintain various invariants. For example, instrumentations that rely on extending existing objects with new properties must take proper care not to leak information that is visible to user code by introspection (e.g., by iterating over all properties of an object[1]).

Both VM instrumentation as well as source-to-source transformations can have unexpected performance costs. VM instrumentation often settles for modifying a simple non-optimizing interpreter to avoid the additional complexity of instrumenting a commercial Just-In-Time (JIT) compiler. The performance hit incurred by disabling the JIT compiler in a modern JS implementation is significant, often an order of magnitude or more. Second, while source-to-source transformations can benefit from the full range of optimizations performed by the JIT, a naive transformation often results in a similar slowdown.

In this paper, we present an alternative technique for run-time monitoring of JS applications based on *virtual machine layering*. Virtual machine layering consists of exposing implementation-level operations performed by the VM through various abstraction layers. Specifically, our approach uses a flexible *object model* as a basis to build the abstraction layers. A JS application is then transformed to make use of these abstractions. Because this transformation is performed during the execution, the resulting framework can be viewed as a metacircular VM written on top of a host VM for the source language. This approach has three main advantages. First, exposing implementation-level operations provides

[1] Marking properties as non-iterable is not sufficient in general, since `Object.getOwnPropertyNames` will return all property names, irrespective of their iterable nature.

a good compromise between the portability offered by source-to-source translations and the expressiveness of VM modifications. For instance, profilers can easily extend or redefine the implementation-level operations to accomplish their specific tasks. Second, by exposing implementation-level operations in a separate layer, our approach can prevent interference between VM code and user code. This is achieved by ensuring that user code only manipulates objects through *proxies*[2], which provide a form of sandboxing over the native objects provided by the host VM. Finally, the metacircular VM can leverage fast operations provided by the underlying host VM to reduce the overhead of the transformation. This is achieved by (i) letting the host VM execute operations for which no abstraction is necessary, and (ii) providing abstractions that use or support the operations that are efficiently implemented by the host VM. Reusing complex primitive operations from the host VM also greatly reduces the development effort required to provide a fully compliant VM implementation.

Virtual machine layering is not new and has been previously studied as an implementation technique for metaobject protocols [7]. It can add to JS many of the functions of an intercession API such as the Java Virtual Machine Tools Interface (JVMTI) by reifying implicit operations of the language. In contrast to JVMTI, it does not require the modification of the internals of a VM, only a single intercession point in the browser to maintain the invariants of the layered VM by translating dynamically loaded code before it is executed by the host VM. However, it cannot give access to implementation-specific information such as garbage collection events or exact memory usage. A standard API would supersede it, but until consensus is reached by VM implementors, VM layering can help build on a common instrumentation infrastructure and explore the API design space. One of the authors wrote a small patch to add an intercession point to the Debugger API in Firefox[3]. We believe it should be straightforward to implement and require little maintenance to support on all major browsers.

Photon[4], our prototype implementation of this technique, uses a single primitive operation, message-sending, to reify implementation-level operations such as object operations and function calls. The use of the message-sending primitive provides a simple and dynamic mechanism to instrument and even redefine the behavior of a reified operation. For instance, a profiler could intercept all calls by providing a wrapper function for Photon's `call` primitive operation. In order to offset the cost of the message-sending mechanism, Photon implements a *send cache* optimization. This optimization allows the behavior of a message send (e.g., a property access) to be specialized at a given program point. This caching optimization is crucial to obtain a good performance in practice, making Photon on average 19% faster than a commercial interpreter while providing a much higher degree of flexibility and dynamism.

This paper makes two main contributions: (i) the design of a VM that reifies object operations and function calls around a single message-sending primitive

[2] We refer to the implementation concept in general, not the upcoming JS Proxies.
[3] `https://bugzilla.mozilla.org/show_bug.cgi?id=884602`
[4] `https://github.com/elavoie/photon-js/tree/ecoop2014`

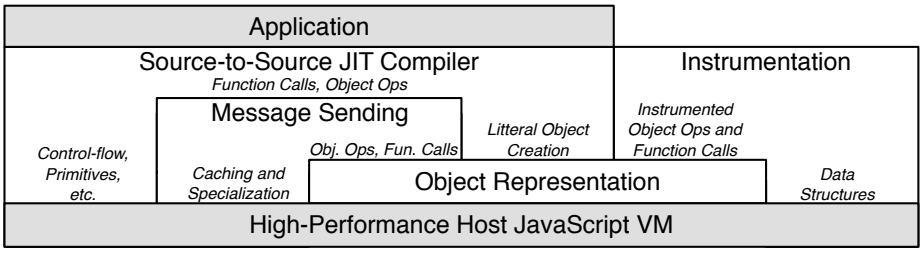

Fig. 1. Components of the Photon virtual machine

so that their behavior can be redefined dynamically, (ii) an object representation exploiting the underlying host VM's inline caches and dynamic object model for performance. Both are shown to provide a significant performance increase over existing approaches and are an important step towards portable and efficient instrumentation frameworks for JavaScript.

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

2 Overview

In a conventional JS setting, an application runs over a high-performance host VM. In the case of a *metacircular* VM, an additional VM layer is inserted between the application and the host VM. This layer can be a *full* or a *differential* implementation. In a full implementation, the metacircular VM provides all functionalities of the source language. In a differential setting, however, the metacircular VM only implements parts of the required functionality, and delegates the remaining operations to the underlying host VM. Our approach follows a differential strategy. Object operations are handled by one of the layers introduced by Photon while primitive operations are handled by the host VM.

This section presents Photon's design goals and components.

2.1 Design goals

Our design aims to achieve the following properties:

- **Isolation**: The application is isolated to avoid any interference with instrumentation code, while still allowing an instrumentation to fully inspect and modify the application state.
- **Abstraction**: Low-level details, mostly related to performance optimizations, are encapsulated to simplify the definition of instrumentations.
- **Performance**: Native features are reused when possible (e.g. control-flow operations). The performance of some host features (e.g., fast global function calls) is leveraged in optimizations that reduce the overhead of abstractions.

In this paper, we focus on the performance aspect to stress the feasibility of virtual machine layering on modern JS VMs.

2.2 Overview of the Components

Figure 1 shows a structural view of the components of Photon.

Source-to-Source Compiler. The source-to-source compiler translates the original JS code to use the runtime environment provided by Photon. Non-reified elements, such as control-flow operations as well as primitive values and operations are preserved. Object operations and function calls are translated to make use of the message sending layer. Literal object creations are translated to use the object representation. The source-to-source compiler is itself written in JS 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.

Message Sending. Photon uses a message sending primitive to reify operations internal to the implementation, such as property accesses on objects and function calls. These reified operations can then easily be overridden and redefined when required, for example to profile the application or to specialize the behavior of an operation. Photon itself makes use of this extra level of indirection for performance by providing a caching mechanism at each site that performs a message send, a form of memoization.

Object Representation. In order to isolate the application from the instrumentation and the host VM, Photon provides a virtualized representation of objects (including functions). Each JS object in the original application is represented in Photon by *two* distinct objects: a property container and a proxy[5]. The property container corresponds to the original object, and acts as storage for all properties that are added to an object. For performance reasons, the property container object is a native JS object provided by the host VM. This allows Photon to leverage its efficient property access mechanism.

The native property container can only be accessed through Photon, and never directly from the application. All object operations go through the proxy object, which is the object that is manipulated directly by the transformed application code. Object representation operations can be specialized in certain classes of objects for performance, such as indexed property accesses on arrays. The use of proxy objects also simplifies the task of implementing instrumentations because it abstracts implementation details that are required for performance. It also allows object-specific instrumentation information to be stored on a proxy without risk of interference with the application properties.

Instrumentation. 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 as opposed to being limited to a specific event before and

[5] Implemented using a regular object. It would be interesting future work to investigate how the upcoming JS proxies perform.

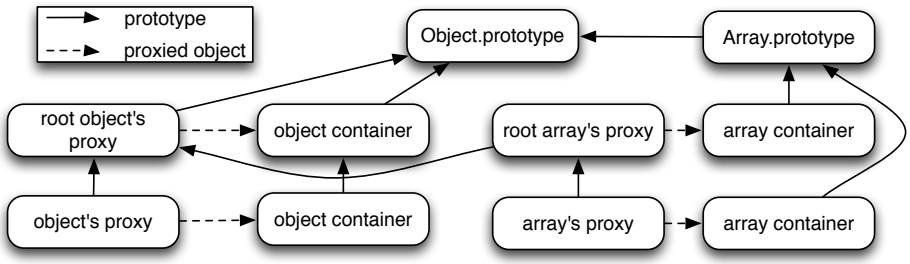

Fig. 2. Representation for objects and arrays

after an operation. However, most instrumentations will choose to simply delegate to the original implementation of an operation and act as wrappers. An instrumentation is executed with the same privileges as the VM, and can therefore directly access the execution environment of the VM. It can also use native objects as data structures.

The next sections expand on the object representation and message sending.

3 Object Representation

Conceptually, all JS objects are associative arrays where the keys represent the *properties* of an object. As with many dynamic languages, properties can be dynamically added, redefined or removed from an object. Each object also has a reference to a *prototype* object from which it inherits properties. The sequence of prototype objects until the root of the object hierarchy forms the *prototype chain* of an object. Functions are also objects, and are first-class citizens in the language. Methods on objects are simply properties with functions as values. JS also treats all global variables and function declarations as properties of a singleton *global object*.

Photon virtualizes the host VM objects exposed to the application in order to provide isolation between the application and the instrumentations, and to reify the object operations supported by JS. While this design provides a high level of flexibility, it also introduces a source of overhead. Proper care must be taken to limit the performance impact of the object representation.

Figure 2 illustrates the object representation used in Photon, with `Object.prototype` as the root of all objects. Photon structures the object representation as proxies to native objects [5]. Each original JS object is therefore represented by two distinct objects in the transformed application. In this representation, the structure of the native (i.e., proxied) object is the same as with the original representation. Using native objects to store properties is motivated by performance. Modern JS VMs aggressively optimize property accesses and method calls on objects, as these operations are key to good performance in

Operation	Interface	Example
Property read	`get(id)`	`o.get("p")`
Property write	`set(id, val)`	`o.set("p",42)`
Property delete	`del(id)`	`o.del("p")`
Prototype read	`getPrototype()`	`o.getPrototype()`
Object creation	`create()`	`parent.create()`
Call	`call(rcv, ..args)`	`fun.call(global)`

Table 1. Object representation operation interfaces

practice. Therefore, preserving the internal structure of the represented objects enables the optimizations performed by the host VM, such as lookup caching.

The application root objects are virtualized for isolation. For example, the application `Object.prototype` is a child of Photon's `Object.prototype`. It is referred to as `root.object` in Photon's implementation. Other JS object model root objects, such as `Array.prototype` are also reified and have `root.object` for prototype.

The proxy object encapsulates the logic implementing the object operations, as well as the invariants that are required for performance (e.g., invalidating caches in response to a redefined operation). Table 1 lists the methods that are provided by proxy objects in order to reify object operations.

Additionally, in order to exploit the fast lookup chain implementations provided by the host VM, the prototype chain of the proxies mirrors the prototype chain of the native objects. This organization of the proxy objects enables specializing and optimizing the operations performed on the object representation at run-time by strategically defining specialized methods along the proxy prototype chain. For example, property accesses performed on array objects can be optimized for the case where the property is numerical rather than using a less efficient, generic mechanism.

However, this strategy does not work well with native types that can be created using a literal syntax, such as arrays, functions and regular expressions. In order to preserve the prototype chain invariant, it would be necessary to change the prototype of these objects after their creation. While technically possible, doing so would invalidate structural invariants assumed by the host VM, at the cost of performance. For such objects, the original native prototype is maintained. When a lookup is needed, it is performed explicitly through the proxy prototype chain. This is illustrated for arrays in the right part of Figure 2.

Although proxies mirror native objects in their prototype chain, they do not mirror their properties. In fact, their properties will be fixed for the whole execution if the object operations are not redefined (e.g., through an instrumentation). Proxies can therefore adapt to dynamic circumstances by adding specialized methods at run-time, which can be used for performance gains. The next subsections demonstrate how this can be exploited to specialize operations for a fixed number of arguments.

3.1 Specialization on a Fixed Number of Arguments

Our object representation does not mandate a specific calling convention for functions. Function calls are reified through a `call` method implemented by function proxies. The naive implementation of `call` uses the equivalent `call` or `apply` method provided by the host VM. However, this generic mechanism is inefficient. It can be avoided by globally rewriting every function to explicitly pass the receiver object. This way, a specialized call operation on a proxy object can simply and efficiently invoke the native function with all arguments passed explicitly. Therefore, function calls can be specialized for the number of arguments found at a given call site. For example, a `call` operation specialized for one argument in addition to its receiver could be implemented as follows:

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

Note that all callable proxies must provide an implementation of `call1` (e.g., by defining this operation on the `FunctionProxy` root).

4 Message-Sending Semantics

Source-level instrumentations aim to intercede on common and often opaque operations performed by the host VM. Our object representation provides a mechanism that reifies implementation-level object operations. In order to enable the redefinition of such operations in a flexible, dynamic and efficient way, our approach uses a single message sending primitive. Translating opaque operations to our message-sending primitive makes them available for instrumentation, and provides additional performance benefits.

4.1 Reifying Object Operations

Reifying opaque operations in source-level instrumentations is typically achieved by transforming the original code so that all such operations go through globally accessible functions. For example, in the case of the property read `var v = o.foo`, the program could be instrumented as follows:

```
function __get__(o, p) {
    <before>
    var r = o[p];
    <after>
    return r;
};
...
var v = __get__(o,"foo");
```

This strategy exposes the details of the opaque operation, such as the identity of the object as well as the name of the property being accessed. It allows an instrumentation to perform some work *before*, *after* or even *instead of* the original operation. However, it lacks flexibility. For instance, instrumentations

requiring a fine-grained control over which objects need to be monitored would need to introduce tests in the global function, at a cost in performance. Also, this rigid design makes it difficult to disable the instrumentation dynamically without incurring the run-time cost introduced with the instrumentation mechanism. Furthermore, multiple optimizations cannot be combined seamlessly without adapting the intercession mechanism.

To address these limitations, our approach replaces globally accessible functions with methods defined on the objects being monitored. This strategy exploits the object-oriented nature of the underlying implementation, and enables a fine-grained monitoring strategy to be implemented easily. For example, an instrumentation of property reads could be implemented as follows:

```
o.__get__ = function (p) {
    <before 1>
    var r = this[p];
    <after 1>
    return r;
};
Array.prototype.__get__ = function (p) {
    <before 2>
    var r = this[p];
    <after 2>
    return r;
};
Object.prototype.__get__ = function (p) {
    <before 3>
    var r = this[p];
    <after 3>
    return r;
};
...
var v = o.__get__("foo");
```

This example illustrates how an instrumentation can be applied selectively to a set of objects based on their hierarchy. This example performs a different instrumentation for three distinct classes of objects: a given instance `o`, all arrays, and all other objects. While there is an added cost to this technique, it preserves the ability of the host VM to optimize the calls to `__get__` using its regular inline caching mechanism.

Note that to ensure isolation, this instrumentation strategy is combined with the object representation presented in Section 3. All operations are therefore performed on proxies instead of accessing the native object directly:

```
proxy.set("__get__",
      new FunctionProxy(function (p) {
          <before 1>
          var r = this.get(p);
          <after 1>
          return r;
      }));
...
function send(proxy, msg, ..args) {
    return proxy.get(msg).call(obj, ..args);
```

Object Model Operation	Example	Equivalent Message Send
Property read	`o.p`	`send(o,"__get__","p")`
Property write	`o.p=42`	`send(o,"__set__","p",42)`
Property delete	`delete o.p`	`send(o,"__del__","p")`
Object creation with literal	`{p:42}`	`send({p:42},"__new__")`
Object creation with constructor	`new C()`	`send(C,"__ctor__")`

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

```
}
var v = send(proxy, "__get__", "foo");
```

The `send` function in the previous example encapsulates the message sending logic as implemented by Photon. The semantics of the `send` operation correspond to a regular method call: the function proxy corresponding to a given message is first looked up, possibly using the prototype chain, and is then invoked with the provided arguments. While this formulation is not strictly necessary to obtain the desired semantics, our current implementation relies on it for performance optimizations, as explained in Section 4.3.

The strategy used to support *__get__* can be used to support all other object operations. A summary of the supported operations and their equivalent message sends is listed in Table 2.

4.2 Reifying Function Calls

JS functions can be called directly (e.g., `f()`) or indirectly through their `call` method. This mechanism can be seen as a form of built-in reification of the calling protocol. However, there is no causal connection between the state of the `call` method and the behavior of function calls: redefining the `call` method on `Function.prototype` does not affect the behavior of call sites. Therefore, `call` is not sufficient to expose all function calls for instrumentation purposes.

This causal relationship is established in our approach by providing a `call` operation on all function proxies. Similarly to other object operations, all function calls in the original program are transformed into a send of the `call` message to a function proxy. Table 3 lists the transformation strategy for each type of function call provided by JS. Note that global function calls are translated directly into method calls on the global object, thereby exposing their semantics at the compilation stage. In order to implement both method calls and regular function calls using the same mechanism, a modification of the `send` operation ensures that the reified `call` operation is used for all calls throughout the system:

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

Call Type	Description	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")`

Table 3. Call types and their equivalent message sends

With these mechanisms in place, all function calls can be instrumented simply by redefining the root function's `call` method.

4.3 Efficient Implementation

In order to reduce the indirection introduced by the transformation process, Photon uses a caching mechanism for send operations. *Send caches* use global function calls both as an optimized calling mechanism as well as operations that can be redefined dynamically. They provide the same ability as code patching in assembly. On the state-of-the-art JS VMs, inlining functions becomes possible when their number of expected arguments matches the number of arguments supplied. If the global function is redefined at a later time, the call site will be deoptimized transparently. This is a highly powerful mechanism because much of the complexity of run-time specialization is performed by the underlying host. The caches implemented by Photon piggyback on this approach.

For example, sending the message `msg` to an object `obj` inside a `foo` function can be written as follows:

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

The `send` function is a global function. It can be replaced with another global function that is guaranteed to be unique, so that each call site effectively receives its own version of the `send` primitive. In addition to the message to be sent, this global function is also provided with a unique identifier used to access the corresponding global function name, for later specialization of the call site:

```
function initialState(rcv, dc, ..args) {
    <<<code updating variable "scN" (N=dc[0])>>>
    return send(rcv, dc[1], ..args);
}

var sc0 = initialState;
var dc0 = [0, "msg"];

function foo(obj) {
    sc0(obj, dc0);
```

```
}
```

Note that the `initialState` function follows the same calling convention as the `send` function. Furthermore, `dc0` can be used to store additional information according to the state of the cache, if needed.

After an initial execution, the cache will dynamically be redefined to hold an optimized version of the operation. For the example, the default caching mechanism implemented by Photon will specialize the cache as follows:

```
var sc0 = function (rcv, dc) {
    return rcv.get("msg").call(rcv);
};
var dc0 = [0, "msg"];

function foo(obj) {
    sc0(obj, dc0);
}
```

Apart from the indirection of the global function call, this example is optimal with regard to the chosen object representation. If the underlying host VM chooses to inline the global function, the cost of the indirection will be effectively eliminated in practice.

In addition to the inlining of the message sending operation in terms of the object operations, as shown previously, Photon also uses the cache to avoid the cost of message sending altogether for reified operations, by inlining an optimized version of its behaviour. In this case, the reified operation is assumed to be defined only once on the root object. Photon tests it by looking for a `__memoize__` property on the method (explained in the next subsection).

That limitation is necessary because, when an instrumentation redefines the reified operation simultaneously on more than one object, Photon's current invariant tracking mechanism cannot detect whether the instrumented method of the current receiver object would resolve to the one inlined. It is assumed, in this case, that an instrumentation writer would not define a `__memoize__` property on the instrumented operation in order to prevent the application of that second optimization.

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 define a memoized method in our approach to be a method that performs the same operation, albeit possibly more efficiently by exploiting run-time information (e.g., argument count). This particular functionality is necessary to efficiently implement the JS object operations in our system because they are reified as methods.

The basic principle behind memoizing methods is to allow a method to inspect its arguments and receiver in order 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 defining 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 there is no penalty for temporarily redefining the calling method after the original method has been restored.

Specializing instrumentations Performance-critical instrumentations can use memoization to provide efficient specialized operations. For example, consider a simple instrumentation that counts the number of property accesses:

```
root.object.set("__get__",
                new FunctionProxy(
                    function ($this, prop) {
                        counter++;
                        return $this.get(prop);
                    }));
```

The redefinition of the *__get__* operation prevents the use of the default inlining mechanism, and therefore reverts the send cache behavior to the following:

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

var sc0 = function (rcv, msg, prop) {
    return rcv.get("__get__").call(rcv, prop);
};

sc0(o, "__get__", "p");
```

To limit the incurred performance overhead, this instrumentation could provide an implementation of *__get__* that additionally responds to the `__memoize__` message. After the first execution of the property access, the optimized version of the send cache would become specialized as follows, thereby eliminating much of the additional overhead from the naive implementation:

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

var sc0 = function (rcv, msg, prop) {
    counter++;
    return rcv.get(prop);
};

sc0(o, "__get__", "p");
```

Cache States and Transitions In order to guarantee the correct behavior of an application, caches need to be invalidated when their invariants are violated. This requires tracking the invariants for each cache used in the system. To simplify tracking the invariants, we always perform lookups for method calls (i.e.,

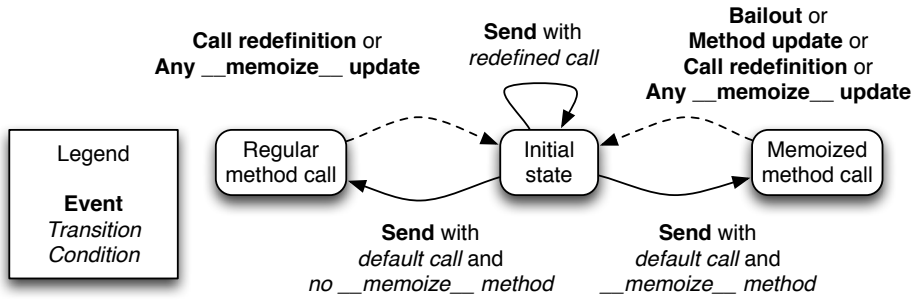

Fig. 3. Cache States and Transitions

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.

In addition to its *initial state*, each cache can be in one of two states, *regular method call*, in which the method is first looked up and called, and *Memoized method call*, in which a method-specific behavior is executed.

Transitions between states happen on message-sends and object-operation events. We choose to 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. Memory usage is proportional to the number of active 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 3 summarizes those elements in a state diagram. A more detailed explanation of every event and transition conditions is given in Table 4.

Our current tracking strategy was chosen to evaluate the performance of our prototype with a minimal implementation effort. However, it is not granular enough to track instrumentations redefining operations on non-root objects. A more granular strategy should be used for instrumentations requiring different operations for different groups of objects.

5 Compilation and Execution Example

We now show how the components of Photon work together using an example. It illustrates many of the reified operations discussed previously: property reads and writes as well as function and method calls. Consider the following program:

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.

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.

Table 4. Cache Events and Transition Conditions

```
var f = function (n, d) {

  for (var i=1; i<=2; i=i+1) {
    n = n + d.getTime();
  }

  return n;
};

f(42, new Date(100));
```

Note that the `getTime` method call will be executed twice during execution.

The source-to-source compiler translates each reified operation to a message send according to Table 2 and Table 3. Each occurrence of a message send has an associated send cache (`sc`n) initialized to the `initialState` function, and a data cache (`dc`n), containing the cache identifier (n), the message name and compile time information about arguments. Each literal object created is wrapped in a proxy to obey the object representation, a function literal is therefore wrapped with a `FunctionProxy`. Non-reified operations, such as the scope chain accesses, control-flow operations, such as the **`for`** statement, numbers and arithmetic operations are preserved as-is in their original form.

The commented original code is weaved with the generated code for clarity:

```
sc1 = initialState; // SC for: var f = ...
dc1 = [1,"__set__",["ref","string","scSend"]];

sc2 = initialState; // SC for: function (n,d)...
dc2 = [2,"__new__",[]];

sc3 = initialState; // SC for: d.getTime()
```

```
dc3 = [3,"getTime",["get"]];

sc4 = initialState; // SC for: f(42, ...)
dc4 = [4,"f",["ref","number","scSend"]];

sc5 = initialState; // SC for: new Date(100)
dc5 = [5,"__ctor__",["scSend","number"]];

sc6 = initialState; // SC for: Date
dc6 = [6,"__get__",["ref","string"]];

sc1(root_global,          // var f =
    dc1,
    "f",
    sc2(root.func,        // function (n,d) {
        dc2,
        new FunctionProxy(
              function ($this,n,d) {
                var i = undefined;
                for (i=1; i<=2; i=i+1) {
                  // n = n + d.getTime();
                  n = n + sc3(d, dc3);
                }
                return n;
              })));       // };

sc4(root_global,          // f(42,
    dc4,
    42,
    sc5(sc6(root_global,  //   new Date(100));
            dc6,
            "Date"),
        dc5,
        100));
```

When executed, this code will perform message sends at each of the send caches. The third send cache (`sc3`) will benefit from the caching mechanism. The first time around the loop, the `initialState` function in the Photon runtime will be called. Since `getTime` is a regular method call, Photon's runtime will inline the send semantics and specialize it for the number of arguments at `sc3`'s call site by storing a specialized function in `sc3`, equivalent to:

```
sc3 = function ($this) {
    return $this.get("getTime").call0($this);
};
```

Further calls will be made to this function rather than to `initialState`.

6 Performance

Currently there is no general purpose instrumentation framework that has been shown to work on a wide-array of web applications, across browsers, and at a reasonable performance cost. The task of porting to multiple browsers and supporting the fast evolution of web standards is beyond the capacity of a small

research team, and we did not attempt it. The rest of this performance evaluation should be read in that light.

Nonetheless, our work on Photon has produced interesting performance results. When compared to the slowdowns observed on other systems, they suggest the approach helps reduce the perceivable latency on instrumented applications.

We identified interpreter-level performance as the target because from private communications with other researchers and anecdotal evidence from published work [11,10], this is what typically ends up being instrumented in practice, without any portability across browsers or browser versions. That level of performance is reported to be "barely noticeable on most sites" [11]. Our approach provides a similar performance while being portable.

Our evaluation shows that Photon is portable across many popular browser VMs and that it is faster than other published systems.

6.1 Setting

We chose CPU-bound benchmarks, which although not representative of typical web applications [10], represent the worst-case in terms of instrumentation overhead. For this reason we have mainly used the V8 benchmark suite version 7 in our performance evaluation. These benchmarks are self-checking to detect execution errors. We ran the benchmarks five times and took the average.

To investigate portability, we have used four different JS VMs in our experiments: three VMs based on JIT compilers and one VM based on an interpreter. The following web browsers were used:

- **Safari** version 6.0.2 (8536.26.17), which is based on the Nitro JS VM.
- **Chrome** version 25.0.1364.172, which is based on the V8 JS VM.
- **Firefox** version 20.0, which is based on the SpiderMonkey JS VM. Firefox was run with the JIT enabled, and also with the JIT disabled (which causes the SpiderMonkey interpreter to be used). To disable the JIT we have set the following Firefox javascript options to false, as suggested by the SpiderMonkey development team: `ion.content`, `methodjit.chrome`, `methodjit.content`, `typeinference`. Note that disabling SpiderMonkey's type inference actually accelerates the execution of all programs because the interpreter does not take advantage of the type information.

Chrome does not have an interpreter and recently, the Safari interpreter was rewritten in an assembly language dialect for performance, making its modification for instrumentation more complicated. We therefore think that the only remaining interpreter that is both simple and fast enough for instrumentation is the Firefox interpreter.

To simplify the description of the results, we will conflate the name of the web browser with that of its JS VM.

A computer with a 2.6 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM and running OS X 10.8.2 is used in all the experiments.

The experiments can be run by visiting the corresponding links from the project web page[6]. Individual results are reported as well as average value, standard-deviation, and ratios between configurations.

6.2 Related systems

To put the performance results we obtained in context, we compared against alternatives. Either they ran fewer of the V8 benchmarks than Photon, they had a higher slowdown or both. The related work section compares them with Photon in more details.

Js.js [13] is a JS port of the Firefox interpreter compiled using the Emscripten C++ to JS compiler. This is a heavy-weight approach with a significant performance overhead and, presumably, would require a similar amount of effort to instrument as if the Firefox interpreter was instrumented. The EarleyBoyer and Splay benchmarks ran out of memory, RayTrace crashed the version of Chrome we were using and RegExp would trigger a malloc error in Js.js. NavierStokes would take more than 10 minutes to complete and the other benchmarks would show slowdowns between 5243× and greater than 18515×.

Jalangi [12] is a record-replay and dynamic analysis framework for JavaScript. We independently tested their system using V8 benchmarks using the preconfigured virtual machine[7] they provide on their website and found their system to introduce slowdowns between 384× and 2520× during recording, except for RegExp (15×) and Splay (29×). We also verified that on some of the interactive application they used for testing, the slowdown was noticeable but not to the point of completely hindering the interaction. The main take away is that although some of the slowdowns on CPU-bound benchmarks may seem impressive, the additional latency is acceptable in practice.

AspectScript [14] is similar to Photon but uses the aspect formalism as an interface for designing dynamic analyses. We executed the latest version of AspectScript against the V8 benchmarks, and found it to be between 10× and 454× slower than Photon on Safari. Additionally, only four of the benchmarks ran without errors.

JSProbes [3] and work by Lerner et al. [8] modified the host VM but both are now incompatible with current browser VMs.

Narcissus could run none of the V8 benchmarks and was two orders of magnitude slower than Photon on a micro-benchmark stressing the function calls.

In the next sections, we investigate the performance behaviour of Photon.

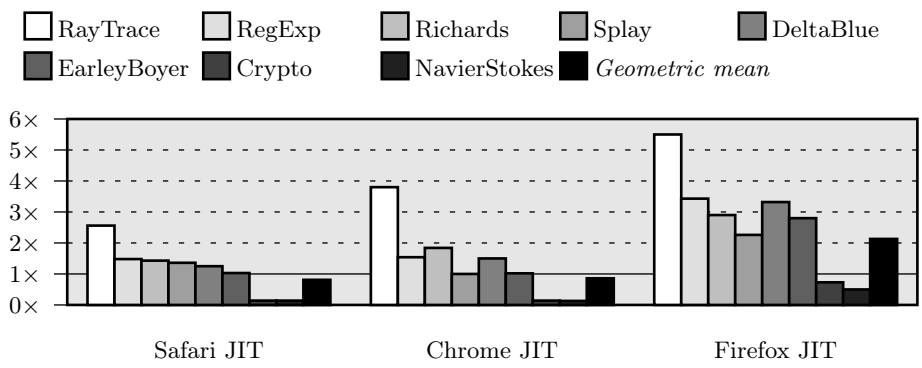

Fig. 4. Relative performance of Photon on various VMs compared to the Firefox interpreter

6.3 Comparison with interpreter instrumentation

Figure 4 gives for each benchmark and JIT VM the execution speed ratio between Photon with no instrumentation and the Firefox interpreter. Therefore, on average, Photon without instrumentation runs the benchmarks faster on Safari JIT (by 19%) and Chrome JIT (by 14%) than when they are run directly on the Firefox interpreter. The execution speed with Photon is consistently faster over all JIT VMs for Crypto and NavierStokes which run about 7× faster with Photon on Safari JIT and Chrome JIT. The major increase in performance can be attributed to a substantial proportion of the time spent in features that are not instrumented by Photon, either native libraries or language features other than object operations and function calls. The worst case for all JIT VMs occurs for RayTrace, which is 2.5× to 5.5× slower when executed with Photon. This shows that the performance of Photon running on a JIT VM is roughly in the same ballpark as an interpreter.

6.4 Inherent overhead compared to JIT compilation

Figure 5 shows the slowdown caused by Photon on each VM relative to executing the program without Photon on the same VM. These results mostly show that (1) selective program transformation can benefit from the native performance of features that are not instrumented, that (2) performance is not portable across browsers, given the significant variability in performance results observed on the same benchmarks between browsers, and (3) the interpreter is much less affected

[6] `http://elavoie.github.io/photon-js/`

[7] On OS X 10.8.5, on an 1.8GHz Intel Core i7 with 4 GB of RAM, with the virtual image running LUbuntu 13.04 with 2GB of RAM and Jalangi commit 5f6d538d9e.... The virtualization was found to introduce a 15% slowdown compared to running the V8 benchmarks on the host.

by program transformation than JITs are, which suggests that the peformance of JITs is highly dependent on the nature of the code.

Newer experiments[8] show the maximum memory usage ratios to be between 1.25× and 4.0× with EarleyBoyer(27.7/7.2MB) and Splay(164/92.5MB) being the relative and absolute worst. The absolute memory usage suggests Photon is practical on current desktop and laptop machines.

They also show that the host VM inline caching is crucial for Photon's performance and execution. Disabling inline caches on Chrome[9] slows down Photon by an additional factor between 9× and 156× and prevents EarleyBoyer from completing because it runs out of memory.

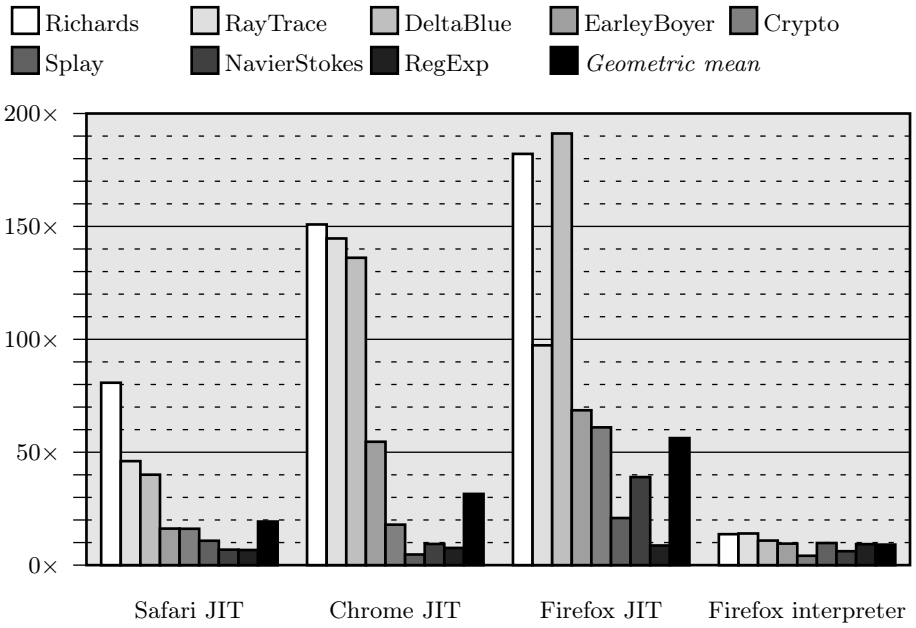

Fig. 5. Inherent overhead (factor slowdown) of Photon on various VMs

6.5 Effect of send caching

For all benchmarks, at the end of execution, all caches that were executed at least once are in one of the two optimized cases and all reified operations are in a memoized state. Deactivating the optimization by performing the method

[8] On OS X 10.8.5, on an 1.8GHz Intel Core i7 with 4 GB of RAM running Chrome version (33.0.1750.117), because the old setup was not available anymore.

[9] By starting it with the –js-flags="–nouse_ic" option.

	Safari JIT		Chrome JIT		Firefox JIT		Firefox interp.	
Benchmark	simple	fast	simple	fast	simple	fast	simple	fast
Richards	2.31×	1.06×	2.38×	1.26×	2.81×	1.07×	1.88×	1.24×
RayTrace	1.59×	1.07×	1.30×	.93×	2.19×	1.02×	1.55×	1.15×
DeltaBlue	2.68×	1.11×	3.16×	1.01×	2.03×	1.02×	1.98×	1.19×
EarleyBoyer	2.18×	1.12×	2.31×	1.14×	2.71×	1.07×	1.78×	1.15×
Crypto	16.80×	1.23×	18.53×	1.00×	6.91×	1.00×	4.33×	1.30×
Splay	1.70×	1.68×	2.45×	1.37×	1.96×	1.05×	1.42×	1.17×
NavierStokes	29.17×	1.07×	39.41×	2.05×	11.86×	1.11×	5.65×	1.36×
RegExp	1.37×	1.01×	1.31×	.99×	1.29×	1.02×	1.30×	1.03×
Geom. mean	*3.54*×	*1.15*×	*3.90*×	*1.18*×	*3.03*×	*1.04*×	*2.15*×	*1.19*×

Table 5. Execution speed slowdown of Photon with a simple and a fast instrumentation of property read, write and delete

lookups on each operation slows down Photon by a factor between 29× and 320× in addition to the previously reported slowdowns, and prevents EarleyBoyer from running because of a stack overflow.

6.6 Performance with instrumentation

We have evaluated the performance of Photon with an instrumentation that counts the number of run-time occurrences of the following object representation operations: property read, write and deletion. We chose this particular instrumentation because it is simple, it covers frequently used object model operations and it was actually used to gather information about JS (it can be used to reproduce the object read, write and deletion proportion figure from [10]).

Two implementations of this instrumentation were used; a simple (~16 lines of code) and a fast version (~100 lines of code)[10]. The simple version does not exploit memoization and corresponds to the straightforward implementation: incrementing a counter and calling the corresponding object representation operation. The fast version uses the memoization protocol to inline the counter incrementations inside the optimized version of the object operations.

The execution speed slowdown of Photon with each version of the instrumentation for each JS VM is given in Table 5. This means that on Safari JIT and Chrome JIT, on average, the benchmarks run with the fast version of the instrumentation on Photon essentially at the same speed as the uninstrumented original benchmarks directly on the Firefox interpreter, while in many cases the simple version is sufficient to obtain a reasonable performance.

7 Limitations

Due to our implementation of the prototype chain, accessing the `__proto__` property leaks the internal representation. This can be solved at a substantial per-

[10] `https://github.com/elavoie/photon-js/tree/ecoop2014/instrumentations`

formance 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, even if very unlikely in practice, render it unsound.

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 unlikely 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 don't think it is worth handling.

Functions passed to the standard library are wrapped to remove the extra arguments introduced by our compilation strategy. However, the wrappers do not perform message sends, therefore these calls are invisible to an instrumentation.

Photon objects cannot be manipulated outside of Photon, the execution environment (e.g. DOM) needs to be virtualized. For the DOM, Andreas Gal's implementation in JavaScript seems a good starting point[11].

8 Related Work

The layering of a metacircular implementation implementing reflection techniques with an object-oriented approach was beautifully explained in "The Art of the Metaobject Protocol" [7]. This paper revisited those ideas while considering the performance behavior of modern JS VMs.

Sandboxing frameworks for JS, such as Google Caja [1], BrowserShield [9] and ADSafe [2] guarantee that guest JS code cannot modify the host JS environment outside of a permitted policy. We focus here on Google Caja as a representative candidate. The Caja sandbox provides a different global object to the guest code and performs a source-to-source translation to ensure that all operations on host objects are mediated by proxies enforcing a user-defined security policy. Photon also provide a different global object for the purpose of simplifying reasoning about instrumentations while providing an acceptable level of performance. Our sandboxing strategy does not need to be as stringent, therefore we deem acceptable the possibility of leaking the native objects by accessing the `__proto__` property.

JSBench [11] performs instrumentation of object operations and function calls for recording execution traces of web applications that can be replayed as

[11] `https://github.com/andreasgal/dom.js`

stand-alone benchmarks. JSBench instrumentation is specially tailored to the task of recording benchmarks while Photon aims to be a general framework.

The idea of using aspect-oriented programming for profiling tasks has been explored in the past, although some limitations of the model have been identified (e.g., [4]). AspectScript [14] has similar aims as Photon, namely providing for JS a general interface for dynamic instrumentation of object operations and function calls. It uses a source-to-source translation scheme with a single *reifier* primitive which is analogous to our *message-sending* primitive. Compared to our instrumentation interface, they use the dynamic weaving of aspect formalism instead of our "operations as methods" approach. Because of the use of the aspects formalism, their approach provides better encapsulation of the instrumentation strategy at the expense of flexibility and performance.

Js.js [13] is a JS port of the Firefox interpreter compiled using the Emscriptem C++ to JS compiler. It is intended for sandboxing web applications. The resulting JS interpreter then runs in the browser on top of an existing VM. Photon avoids reimplementing features other than object operations and function calls. The resulting implementation is both faster and simpler to instrument.

Other approaches target the host VM for efficiency reasons. JSProbes [3] is a series of patches to the Firefox interpreter that allow instrumentations to be written in JS and target pre-defined probe points, such as object creation, function calls and implementation events such as garbage collector start and stop events. JSProbes provides much of the same properties as Photon at a much lower execution overhead and with additional information about implementation events that are inaccessible to Photon. At the time of writing, maintenance of JSProbes has stopped, making the approach unavailable in practice. In a different setting, Lerner et al. explored the requirement for implementing aspect support in an experimental JIT-compiler [8]. They reported a simpler and more efficient implementation than other aspect-oriented approaches. Their work was intended to inform possible ways to open native implementations to instrumentation with an aspect formalism. So far, no production VM implements aspects, which makes this approach unavailable in practice. Photon does not require modifications to the host VM. It therefore does not add to the maintenance cost of production VMs to be usable in practice.

Jalangi [12] is a record-replay and dynamic analysis framework for JavaScript. It performs an ahead-of-time (static) source-to-source translation of the program to replace instrumented operations with function calls. The instrumented program is executed to record a trace of execution, which is then used to perform dynamic analyses. Being static, their translation strategy cannot handle the dynamic aliasing of `eval` or `Object.create`. Photon however, by virtualizing the execution environment, provides wrapper around these methods, which supports dynamic aliasing.

Narcissus JS in JS interpreter implementation by Mozilla that reifies all the language operations of the language. However, compared to Photon, Narcissus is much slower and none of the V8 benchmarks could be executed.

9 Conclusion and Future Work

Run-time monitoring of JS applications is crucial to obtain empirical data about current web applications, to improve their efficiency and improve VM technologies. Unfortunately, there is no general purpose instrumentation framework that has been shown to work on a wide-array of web applications, across browsers, and at a reasonable performance cost. Existing approaches have either modified browser VMs at the expense of portability, or relied on source-to-source transformations that are complex to develop and still incur a significant overhead.

In this paper, we explored the performance aspects of *virtual machine layering*, in which a portable implementation of a JS implementation exposes implementation-level operations that can be redefined at run time to monitor the application execution. We have shown that by a selective dynamic translation of source elements, combined with run-time feedback to optimize the reified operations, we could obtain significantly better performance than existing approaches when exposing object operations and function calls to the point where the approach can be competitive with the instrumentation of a browser interpreter while being portable across VM implementations.

The major challenge remaining, which prevents the application of the approach in practice, is the full and efficient virtualization of the execution environment, whether it is the browser libraries such as the Document Object Model (DOM) or the extensions of the NodeJS framework. One requirement is the possibility of full intercession of all the code loaded. Work is under way to extend Debugger API in Firefox based on the work done at Mozilla by one of the authors to support it. The other requirement is the proper wrapping of all the environment libraries, which is a significant engineering effort but could be reusable for different implementation strategies of virtual machine layering and instrumentation APIs. This could be tackled as a community effort.

10 Acknowledgement

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.

References

1. Google Caja. `http://code.google.com/p/google-caja/` (December 2012)
2. ADSafe. `http://www.adsafe.org/` (March 2013)
3. JSProbes. `http://brrian.tumblr.com/search/jsprobes` (March 2013)
4. Binder, W., Ansaloni, D., Villazón, A., Moret, P.: Flexible and efficient profiling with aspect-oriented programming. Concurrency and Computation: Practice and Experience 23(15), 1749–1773 (2011)
5. Bracha, G., Ungar, D.: 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. pp. 331–344. ACM, New York, NY, USA (2004)

6. Heidegger, P., Bieniusa, A., Thiemann, P.: Access permission contracts for scripting languages. In: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 111–122. POPL '12, ACM, New York, NY, USA (2012)
7. Kiczales, G., Rivieres, J.D.: The Art of the Metaobject Protocol. MIT Press, Cambridge, MA, USA (1991)
8. Lerner, B.S., Venter, H., Grossman, D.: Supporting dynamic, third-party code customizations in JavaScript using aspects. In: Proceedings of the ACM international conference on Object oriented programming systems languages and applications. pp. 361–376. OOPSLA '10, ACM, New York, NY, USA (2010)
9. Reis, C., Dunagan, J., Wang, H.J., Dubrovsky, O., Esmeir, S.: BrowserShield: Vulnerability-driven filtering of dynamic HTML. ACM Trans. Web 1(3) (Sep 2007)
10. Richards, G., Lebresne, S., Burg, B., Vitek, J.: An analysis of the dynamic behavior of JavaScript programs. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. pp. 1–12. ACM (2010)
11. Richards, G., Gal, A., Eich, B., Vitek, J.: Automated construction of JavaScript benchmarks. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications. pp. 677–694. OOPSLA '11, ACM, New York, NY, USA (2011)
12. Sen, K., Kalasapur, S., Brutch, T., Gibbs, S.: Jalangi: A selective record-replay and dynamic analysis framework for javascript. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. pp. 488–498. ESEC/FSE 2013, ACM, New York, NY, USA (2013), http://doi.acm.org/http://dx.doi.org/10.1145/2491411.2491447
13. Terrace, J., Beard, S.R., Katta, N.P.K.: JavaScript in JavaScript (js.js): sandboxing third-party scripts. In: Proceedings of the 3rd USENIX conference on Web Application Development. pp. 9–9. WebApps'12, USENIX Association, Berkeley, CA, USA (2012)
14. Toledo, R., Leger, P., Tanter, E.: AspectScript: expressive aspects for the web. In: Proceedings of the 9th International Conference on Aspect-Oriented Software Development. pp. 13–24. AOSD '10, ACM, New York, NY, USA (2010)

A Artifact Description

Authors of the artifact. Erick Lavoie

Summary. The artifact comprises both Photon, the layered virtual machine used for dynamic program analysis described in the previous paper, and the performance experiments used to obtain the performance figures. The current implementation of Photon initially performs a source-to-source translation of JavaScript code, while running over NodeJS. The resulting code then runs in the browser in a virtualized environment that abstracts the standard libraries and also includes Photon, for correct translation of dynamically generated code. The experiments come packaged as ready-to-run web pages for easy comparison of performance results with newer configurations of browser and machines.

Content. The artifact package includes:

- a set of experiments packaged as ready-to-run web pages;
- the Photon system;
- detailed instructions for using the artifact and running the experiments, provided as an `index.html` file.

To simplify repeatability of our experiments, we provide a VirtualBox disk image containing a Ubuntu Linux image fully configured for testing Photon.

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of Photon code is available on GitHub at `https://github.com/elavoie/photon-js` and a copy of the `index.html` page, including all the performance experiments, is available at the corresponding GitHub page `http://elavoie.github.io/photon-js/`.

Tested platforms. The artifact is known to work on any platform running Oracle VirtualBox version 4 (`https://www.virtualbox.org/`) with at least 5 GB or free space on disk and at least 1 GB of free space in RAM.

License. MIT Licence

MD5 sum of the artifact. 7d38dddb53c801fff254123f45074144

Size of the artifact. 1.03 GB