Dynamic Property Caches

A Step towards Faster JavaScript Proxy Objects

Manuel Serrano
Inria/Université Côte d'Azur
Sophia Antipolis, France
Manuel.Serrano@inria.fr

Robert Bruce Findler
Northwestern University
Evanston, Illinois, USA
robby@cs.northwestern.edu

Abstract

Inline caches and hidden classes are two essential components for closing the performance gap between static languages such as Java, Scheme, or ML and dynamic languages such as JavaScript or Python. They rely on the observation that for a particular object access located at a particular point of the program, the shapes, usually referred to as *the hidden classes*, of accessed objects are likely to be the same. Taking benefit of that invariant, they replace the expensive lookup the semantics of these languages normally demand with one test, the *inline cache*, and a memory read indexed by an offset computed during the last cache miss. These optimizations are essential but they are not general enough to cope with JavaScript's proxies. In particular, when the property name is itself unknown statically, inline cache-based optimizations always take a slow path.

In this paper, we show how to generalize inline caches to cope with an unknown property name. The paper first discusses the general principle of the extension and then presents the experimental results we collected using a modified version of the Hop JavaScript compiler, demonstrating how the optimization is crucial for improving the performance of proxy objects (as they naturally use dynamic property names extensively). The evaluation report shows that the modified Hop outperforms all other implementations of the language, including the most efficient commercial ones, by a factor ranging from 2× to 100×. Even better, our optimizations are applicable to existing compilers as they require only straightforward changes to runtime data structures; no complex analyses are required.

CCS Concepts • **Software and its engineering → Polymorphism**; **Compilers**; **Runtime environments**; *Object oriented languages*; *Classes and objects.*

Keywords JavaScript, Hidden Classes, Inline Caches

CC '20, February 22–23, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7120-9/20/02…$15.00
https://doi.org/10.1145/3377555.3377888

ACM Reference Format:
Manuel Serrano and Robert Bruce Findler. 2020. Dynamic Property Caches : A Step towards Faster JavaScript Proxy Objects. In *Proceedings of the 29th International Conference on Compiler Construction (CC '20), February 22–23, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3377555.3377888

1 Introduction

JavaScript object properties are referred to by their names, which are strings. The canonical syntactic form for property read and property write is `obj[prop]`, where both `obj` and `prop` are expressions. When the string forming the property name is syntactically well formed as an identifier, the access can be abbreviated `obj.prop` (where `prop` is not an evaluated expression when using the dot notation). The semantics of object accesses is precisely defined in the language specification [ECMA International 2011] in an algorithmic manner. For evaluating an expression `obj[prop]` the following steps are executed:

1. evaluate `obj` and convert it into an object if needed;
2. evaluate `prop` and convert it into a string if needed;
3. look up the property in the object and, if not found, repeat the search along the object's prototype chain.
4. produce `undefined` if the property's value is not found.

These steps also apply to array accesses. For instance, assuming that `a` is an array, evaluating the expression `a[10]` requires converting the number `10` into the string `"10"` and then looking for a property of that name in `a` (and possibly its prototype chain). For the sake of portability, the string conversion is precisely defined. For instance, the specification gives the number of decimal digits that must be taken into account for the number conversion.

All fast JavaScript implementations work hard to avoid executing the steps in the semantics literally. For instance, when accessing an array they avoid the number-to-string conversion as much as possible and for accessing an object they use the well known technique of *hidden classes* and *inline caches* to eliminate the lookup in the object and its prototype chain [Chambers and Ungar 1989; Chambers et al. 1989; Deutsch and Schiffman 1984]. Many descriptions of these techniques are available [Artoul 2015; Bruni 2017; Deutsch and Schiffman 1984; Google 2018; Thompson 2015] so there is no need for yet another comprehensive description here. For completeness sake, however, we include an excerpt of

Serrano and Feeley [2019]'s description that shows the basic technique to help present this paper's optimization.

A property access `obj.prop` or `obj["prop"]` can be implemented as follows, using C as the implementation language:

```
(obj->hclass == cache.hclass
   ? obj->elements[ cache.index ]; // cache hit
   : cacheReadMiss( obj, "prop", &cache )) // cache miss
```

On a cache miss, the `cache`'s `hclass` attribute is updated with the object's hidden class. The object's hidden class is updated each time a property is added or removed so the condition `obj->hclass == cache->hclass` holds only for objects that have exactly the same structure. Compared to a structure field access `obj->prop` in C, the overhead of a cache hit is three memory reads (`obj->hclass`, `cache.hclass`, and `cache.index`) and one comparison. All fast JavaScript implementation use similar sequences for accessing object properties.

Various improvements to inline caching have been studied (notably "polymorphic inline caching" [Hölzle et al. 1991; Serrano and Feeley 2019]), but these techniques assume that the property names themselves are invariant. This limitation is visible in the code above, as it has a single use of `"prop"` in the cache-miss code. Optimizing situations where the name is dynamic are the subject of this study.

Dynamic property names are not as common as static ones, so the optimization we propose in this paper is not as critical as the inline cache optimization. That said, dynamic property names are used by the `for..in` construct (a construct that dates to the first version of JavaScript) and, more importantly, they are used extensively with proxy objects (introduced in ECMAScript 6 [ECMA International 2015]). While both are important, improving the performance of proxy object is the main motivation for this work.

A proxy object [Van Cutsem and Miller 2010, 2013] encapsulates another object and interposes on the primitive operations it supports: property reads and writes, function calls, property deletion, etc. Proxy objects serve several purposes including security enforcement [Keil et al. 2015], higher order contracts [Strickland et al. 2012a], and gradual typing [Rastogi et al. 2015]. Unfortunately, the design of proxies makes it difficult to implement them efficiently, that is, with performance comparable to those of plain objects. We have conducted a performance evaluation that shows that the performance penalty imposed by proxies on current, efficient commercial JavaScript implementations is a slowdown of about one or two orders of magnitude, which disqualifies proxies from extensive use.

In this paper, we propose several optimizations that reduce the slowdown significantly. In general, the optimizations we offer in this paper are not sophisticated; their value is that we have identified a set of optimizations that are both easy to implement and are effective.

We have implemented all of them in Hop, an ahead-of-time JavaScript compiler [Serrano 2018]. With these modifications, Hop outperforms all other implementations when proxies are used. On average it reduces the slowdown to at most 10x. This is probably not enough yet for extensive use of proxy objects but this is a first step in the direction of increasing proxy usability and likely makes the difference in some applications.

The paper is organized as follows. In Section 2, we first present the string implementation, a central subsystem in the implementation of dynamic property accesses. In Section 3, we show how we optimize dynamic property accesses. In Section 4 we present how we accommodate the inline cache implementation to minimize the performance penalty of proxy objects. In Section 5 we present the evaluation report. In Section 6 we present the related work and we conclude in Section 7.

2 Strings

JavaScript has two kinds of strings, objects created with the `String` constructor and *string values*. The `String` constructor is actually mostly used as a container for the builtin string methods. It is seldom used in actual programs. In contrast, string values are ubiquitous, in particular because they are used to name object properties. In the rest of this section we focus on string values.

2.1 String Values

JavaScript string values are sequences of 16-bit unsigned integers. They are neither UCS-2 nor UTF-16 strings as the `charAt` method does not interpret characters according to the UCS-2 code points and because there is no interpretation of characters whose encoding spans over two code units. The `length` property of a string value is specified not to give the number of UTF-16 characters composing that string, but the number of 16-bit unsigned integers it contains. Giving up on UTF-16 strings enables a linear access to string characters so it might be sufficient for a JavaScript implementation to support strings as specified with UTF-16 code units. However, for fast interfacing with the operating system and for compactness, it might be sensible to distinguish between strings that can be encoded as 8-bit unsigned integers and those that require 16-bit wide integers. This is what Hop does: ASCII strings are encoded as sequences of 8-bit integers and all other strings are encoded using the UTF-8 schema, which enables a fast interface with the operating system and with foreign languages like Python that also use UTF-8 strings. Most other implementations also use the same dichotomy between 8-bit and 16-bit encoding but generally they use 16-bit integers for non-ASCII strings.

JavaScript string values are constructed when string literals appear in the program text, by `String` methods such as charAt, by regular expression matching, and by concatenation (which is used extensively in JavaScript programs). The Mozilla Developer Network page dedicated to strings [Mozilla 2019b] even suggests using the ‘+’ operator to split long literals. As a consequence, fast JavaScript implementations use ropes as the underlying data structure for strings to support fast concatenation [Boehm et al. 1995; Wikipedia 2019].

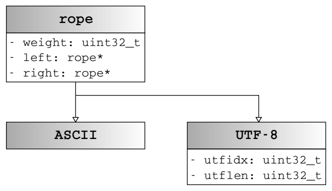

Figure 1. The Hop class hierarchy for JavaScript string values. The `left` and `right` pointers for ASCII strings point only to other ASCII strings. The `left` and `right` pointers of UTF-8 strings point to arbitrary string ropes.

The Hop base class for ropes consists of: a `weight`, a `left` pointer (which is a sequence of characters if the node is a leaf and a node otherwise), and a `right` pointer that might be `null`. A subclass is used to represent ASCII strings. Another subclass is used to represent UTF-8 strings (see Figure 1); the UTF-8 subclass has extra fields to support various unicode optimizations not explained here.

2.2 Property Names

String values are used to name object properties and the property lookup along an object prototype chain compares string values. To optimize this operation, we modified Hop to store string values used to name properties in a global table and implement the property name comparison as a pointer-equality test. More precisely, each string is augmented with an additional `name` field that points to its corresponding unique name. In the evaluation of `obj[prop]`, as described in Section 1, after `prop` is converted into a string value, its associated name is retrieved and used in the operations that follow. The name retrieval is implemented as follows:

```
rope *stringToName( rope *prop ) {
  if( prop->name ) {
     return prop->name;
  } else {
     long hash = stringHash( prop );
     rope *name = globalNameTableGet( hash, prop );
     if( !name ) name = globalNameTablePut( hash, prop );
     prop->name = name;
     return name;
  }
}
```

In spite of its apparent complexity, this function is generally extremely fast. The test `if( prop->name )` succeeds most of the time because, first, string literals are statically allocated as names, *i.e.,* their `name` field points to themselves, so any property name that is explicitly mentioned in the source code does not require any runtime allocation. Second, as soon as a string is used in any property operation, its associated name is allocated once and for all (if it does not already exist) and stored in the string for future use.

3 Dynamic Properties

As presented in the introduction, techniques for implementing efficient object accesses are well understood and deployed in popular compilers. However, they demand that the property name is constant, and thus do not apply when the property name is a dynamic value. In this section, we present the two techniques we have developed to mitigate that problem. The first one applies to array accesses and the second one to regular object accesses.

3.1 Array Property Names

JavaScript arrays are complex to implement efficiently as they support many dynamic operations. They can be dynamically extended or shrunk, they can be sparse, and more importantly, accessing arrays elements obeys the same semantics as object accesses. That is, the index has to be converted into a string value to look it up in the array and, if the index does not have a value, the array’s prototype chain must be consulted.

JavaScript arrays are used so extensively that all implementations try their best to implement them efficiently by using as much as possible flat sequences of values indexed by small integers [Serrano 2018]. Unfortunately, these efficient techniques are not always applicable. A typical pattern that defeats these optimization is the `for..in` loop. Consider this example:

```
for( let i in a ) { s += i + "␣is␣" + a[i] + "␣"; }
```

Let us assume that `a` is an array. The semantics tells us that the local variable `i` must be successively bound to all property names of the array elements, *i.e.,* all the indexes of the elements that the array contains, but where the indices are represented as strings. Thus, in the expression `i + " is " + a[i] + " "` the variable `i` must be a string value. Because of the first argument to `+`, this string is used to form the global result and because of the third (`a[i]`), it is also used to access the array. The semantics thus defeats the simple-minded strategy that represents an array as a linear sequence of memory, for two reasons:

1. Since the array indices are not stored in the object itself, the strings representing the indices must be allocated at runtime.
2. Because `i` must be a string, the conversion to an integer would naturally be handled at runtime. Beyond the cost of parsing the string, the conversion to an integer may even fail as an array may have properties that are not its indicies (*e.g.,* its length).

Problem #1 is similar to the problem of boxing numbers. The compiler folklore tells us that it can be mostly solved by pre-allocating small indexes [Serpette and Serrano 2002]. To avoid the string parsing of problem #2, the second solution is to store, inside the string itself, the corresponding index.

☞ Contribution #1: We extend the class hierarchy of Figure 1 with an additional class for indexes (see Figure 2). In addition to the string itself, instances of that subclass also store the corresponding integer index.[1] Additionally, to avoid dynamic memory allocation for indices, Hop pre-allocates the first N array indices, where N is a global parameter of the system. If needed, the array of pre-allocated indices is extended dynamically.

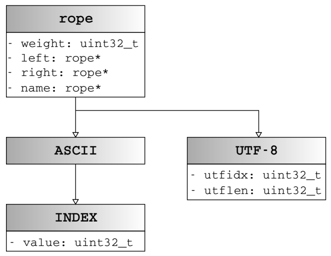

Figure 2. String indices are instances of a dedicated class to avoid expansive string parsing.

This simple framework is memory efficient as indexes are pre-allocated in a memory area that do not even need to be scanned by the garbage collector. It enables optimal conversions from number to string and vice versa. As an index is also a regular rope, strings operations such as concatenation do not need any modification and preserve their efficiency. We compare the performance of this encoding to those used in other systems in Section 5.3 (Figure 10). We show that it enables Hop to be 3× to 5× faster than all other tested systems when proxied arrays are used (see Figure 4).

3.2 Dynamic Property Accesses

When compiling an expression `obj[prop]`, the inline cache technique applies only if `prop` is a constant. A naive attempt to improve the cache would simply convert the given property name and compare it:

```
(( obj->hclass == cache.hclass
   && stringToName( prop ) == cache.prop )
      ? obj->elements[ cache.index ]; // cache hit
      : cacheReadMiss( obj, prop, &cache )) // cache miss
```

but this is ineffective because `prop` is likely to change frequently when it is not a literal string. For instance, consider the `for..in` loop from Section 3.1. Inside the loop, the value of `i` changes at each iteration and thus the naive extension will always miss the cache. The solution we propose is the following:

☞ Contribution #2: *When the property name is dynamic, the cache is attached to the string value itself, instead of to the program location.*

This requires a minor modification to the implementation of string values as they must have extra slots to store the read and write cache. In practice the `right` property and one of the `rcache` or `pcache` can be merged as property names are always normalized ropes, for which only the `left` property is used. In Section 5.2 we evaluate the impact of adding extra fields to the string base implementation. We show that for most tests, the negative impact is invisible. In the worst cases, we have found that it increases memory allocation by 8% and it degrades performance by less than 3% (see Figure 8). Note that regular strings do not need to carry caches. Only strings representing property names do. So, it might be doable to use a more complex class hierarchy than this of Figure 3 where cache attributes are declared only in the subclasses used for property names.

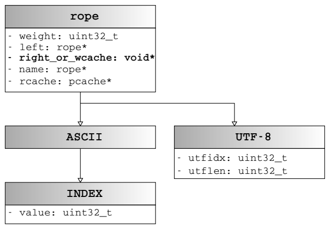

Figure 3. The complete class hierarchy for JavaScript string values with the additional attributes for handling string hashing and string caches. Note that the write cache `wcache` and the rope right-and-side part are merged.

The sequence of accessing a dynamic property is adapted from the conventional inline cache sequence from Section 1:

```
rope *name = stringToName( prop );
if( obj->hclass == name->readcache->hclass ) {
   val = obj->elements[ name->readcache->index ]; // cache hit
} else {
   val = cacheReadMiss( obj, "prop", name ); // cache miss
}
```

We have implemented this technique in Hop and we have evaluated its performance using two micro-benchmarks that are presented in Section 5.3. It establishes that this idea, although simple and easy to deploy, outperforms popular industrial JavaScript implementations. The experimental report also shows that with this modification, Hop is the only system that imposes no significant performance overhead for dynamic property names. We conjecture that is also the main reason why Hop outperforms the other systems by a factor of up to 10× to 100× when testing more general proxy programs (see Section 5.1).

[1]Examining the source code of other engines [Google 2019; Mozilla-central 2019] suggests that they also store the index inside the string, but our performance analysis suggests that the situation is subtle; we explore it in Section 5.3.

4 Proxy Objects

To quote Mozilla's MDN [Mozilla 2019a], a proxy object is used to define custom behavior for fundamental operations (*e.g.,* property lookup, assignment, enumeration, function invocation, etc). It is constructed with: `new Proxy(target, handler)`. The argument `target` is the original object and `handler` is the container for the functions, *a.k.a, traps*, to be invoked when primitive operations are to be executed on `target`. Here is MDN web site proxy example.

```
var handler = {
    get: function(obj, prop) {
        return prop in obj ? obj[prop] : 37;
    }
};
var p = new Proxy({}, handler);
p.a = 1;
console.log(p.a, p.b);
```

Proxy are expected to slow down execution for four reasons:

1. They replace property accesses that are optimized by inline caches with more expensive function calls.
2. They require allocating at least twice as many objects (the original object, the proxy, and, sometimes, the handler), so they exercise the memory allocator and the garbage collector.
3. In the client implementation of traps, target object properties are generally accessed with dynamic property names, as the property name is an argument to the trap. The performance consequences of dynamic property names are severe because the conventional inline cache optimization does not apply.
4. Because both the target *and* the handler can evolve independently from one another, the inline caching should be able to accommodate two independent changes between two accesses to a proxy. Unfortunately, this is not possible with the currently known techniques, the inline caching optimization is effectively defeated for all proxied objects.

Problems #1 and #2 are intrinsic to the very nature of proxy objects. They are unlikely to be eliminated. Problem #3 is mitigated by the technique presented Section 3.2. In the following section we present the solution we propose for improving problem #4.

4.1 General Implementation

Read and write property accesses are the most important operations for proxy performance. Their implementations follow the same principles so we present only one here, namely the write operation. Neglecting error cases for simplicity, the JavaScript standard semantics specify the following operations:

1. check if the proxy object has been revoked.
2. get the handler's `set` property.
3. if `set` is a function, then:
   a. check if the value is compatible with the target.
   b. if it is, invoke the `set` function with three arguments: the target, the property, the value, and handler is the receiver of the method.
4. if `set` is a proxy and if that proxy has an `apply` trap, apply it as step in 3.
5. otherwise, assign the property to target.

☞ **Contribution #3:** We propose techniques for optimizing proxy read and write property accesses. They are evaluated in Section 5.1.

The number of steps and their complexity deter inlining the proxy write sequence inside client code, but we take advantage of the general cache miss sequence to recover the performance. In particular, the proxy sequence is inlined in the `cacheWriteMiss` function.

Previous studies of polymorphic inline caches [Hölzle et al. 1991; Serrano and Feeley 2019] show that they eliminate almost all cache misses for non-proxy programs. But, when a property is read from a proxy object, it always ends up with hidden class comparison failure, as proxy objects do not have properties themselves. This observation justifies that cache misses should favor proxy objects over regular objects (**contrib. #3a**).

Let us now detail how each semantics step is implemented:
Step 1: Hop compiles JavaScript files into an extension of the Scheme language [Kelsey et al. 1998]. JavaScript objects are implemented as Scheme classes, each primitive JavaScript type being mapped to a dedicated class. Hop's Scheme implementation allows instances to store extra information along with the class descriptor at no cost. This is already used by Hop to store information about each object. For instance, one bit is used for denoting inlined arrays, another bit is used for denoting objects that have only regular properties, *i.e.,* read/write/configurable properties, another bit is used to mark sealed object, etc. We used that possibility in our extension for encoding revoked proxies. Testing proxy revocation is then a simple bit comparison (**contrib. #3b**).
Step 2: Getting the `set` attribute of the proxy handler is a normal property access. It can then be optimized using classical inline caches. The critical issue is where and when to allocate the associated cache? The conventional approach, where one cache is allocated per syntactic occurrences of the reference to the `set` attribute, does not work, as every access to any proxy object's `set` attribute has the same source location (as mentioned above, the proxy access sequence is not inlined), namely the part of the runtime system that implements proxies.

Our solution is to have proxies carrying their own caches (**contrib. #3c**). This, however, has its own danger, as it increases the size of proxy objects. Thus, in order to avoid

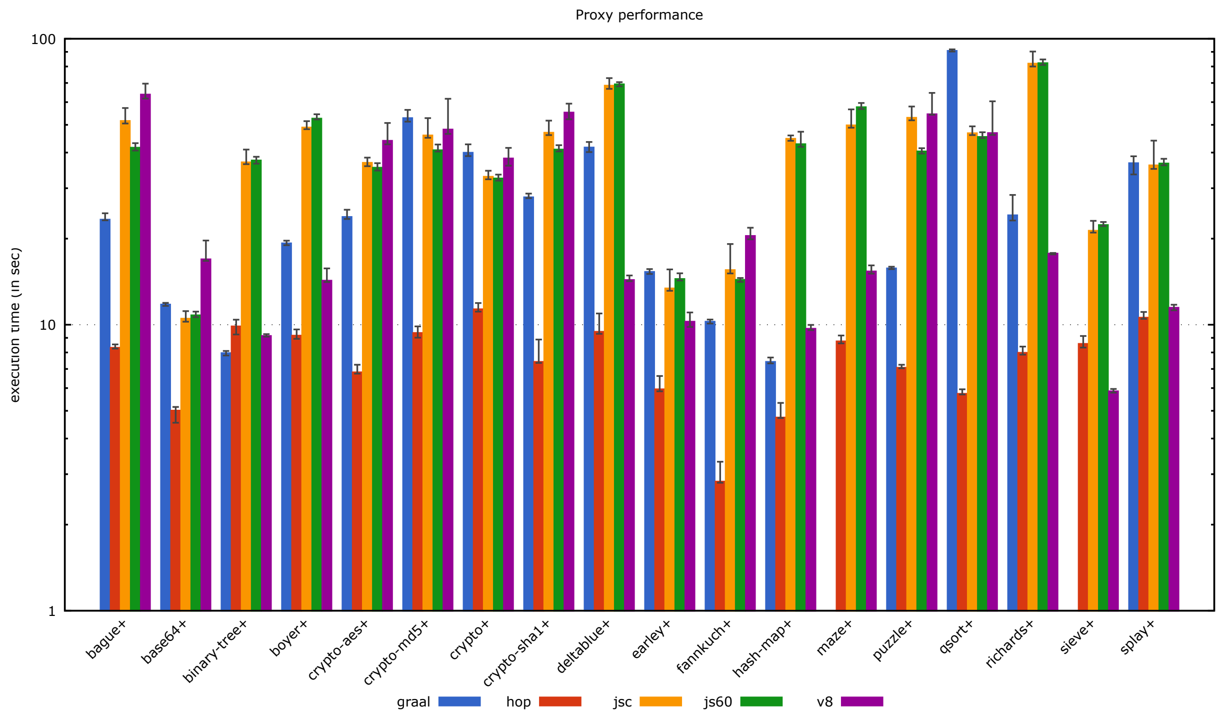

Figure 4. Comparison of proxy performance. Bars represent the mean of 30 execution times for each tested system and for each test. The bars for each implementation are in the same order as in the legend. Smaller is better; log scale.

excessive memory allocation, we allocate shared caches, one cache per proxy allocation site in the original program (**contrib. #3d**). That is, all proxies allocated from the same location will share the same cache for their set and get retrieval. This is generally a good strategy as all proxies allocated on the same site are likely to share their handlers. *Step 3a*: When assigning a value to a proxy object, JavaScript imposes various constraints depending on the definition of the property. For instance, if the property is read-only, the assigned value must be the same as the already stored one. Implementing these checks, in general, require us to retrieve and inspect the property descriptor object of the target [2]. Most objects, however, have (morally) the same property descriptor, namely one that says that all of the properties are read/write. Accordingly, Hop simply has a single bit recorded with the object's runtime representation indicating it has this kind of descriptor. The proxy object runtime support must, therefore, be able to access this bit and also short-circuit the creation of the property descriptor, so the proxy check can be replaced with a check that is a mere arithmetic comparison (**contrib. #3e**).

All the other steps are compiled using standard Hop methods and optimizations.

[2]Each JavaScript object must implement the `getOwnProperty-Descriptor` method that returns per-object description of the object's properties and information about them. Allocating it ahead of time is a significant space cost and implementations generally create it lazily.

5 Experimental Evaluation

A 64-bit Intel Xeon E5-1650 running Linux 4.19/Debian was used for our performance evaluation. Each test is executed 30 times and the median wall clock time with relative standard deviation is collected.

We measure Google's V8 6.8.275.32, JavaScriptCore 4.0 (Jsc), SpiderMonkey C60 (Js60), Oracle's Graal 19.1.1 (Graal), and a modified version of Hop 3.3.0. As all these systems (except Hop) use JIT compilers, we tuned the time of each run to be sufficiently long so that the warm-up time of the JIT is negligible.

We have used different tests depending on the experiment. For testing specific features in isolation, we developed dedicated micro benchmarks. For testing global performance, we use the benchmark suite used for evaluating Hop general performance [Serrano and Feeley 2019]. This test suite excludes some of the classical JavaScript benchmark tests because Hop does not optimize the standard library functions as well as V8, Jsc, and Js60 do. For instance, Hop uses a slow regular engine (pcre based). As a consequence, all the benchmarks that use regexps extensively are biased by that slow library implementation and we have excluded them from hereto presentation. There are similar issues for floating point numbers. Hop uses the Boehm's collector [Boehm and Weiser 1988], a non-copying collector that allocates and deallocates

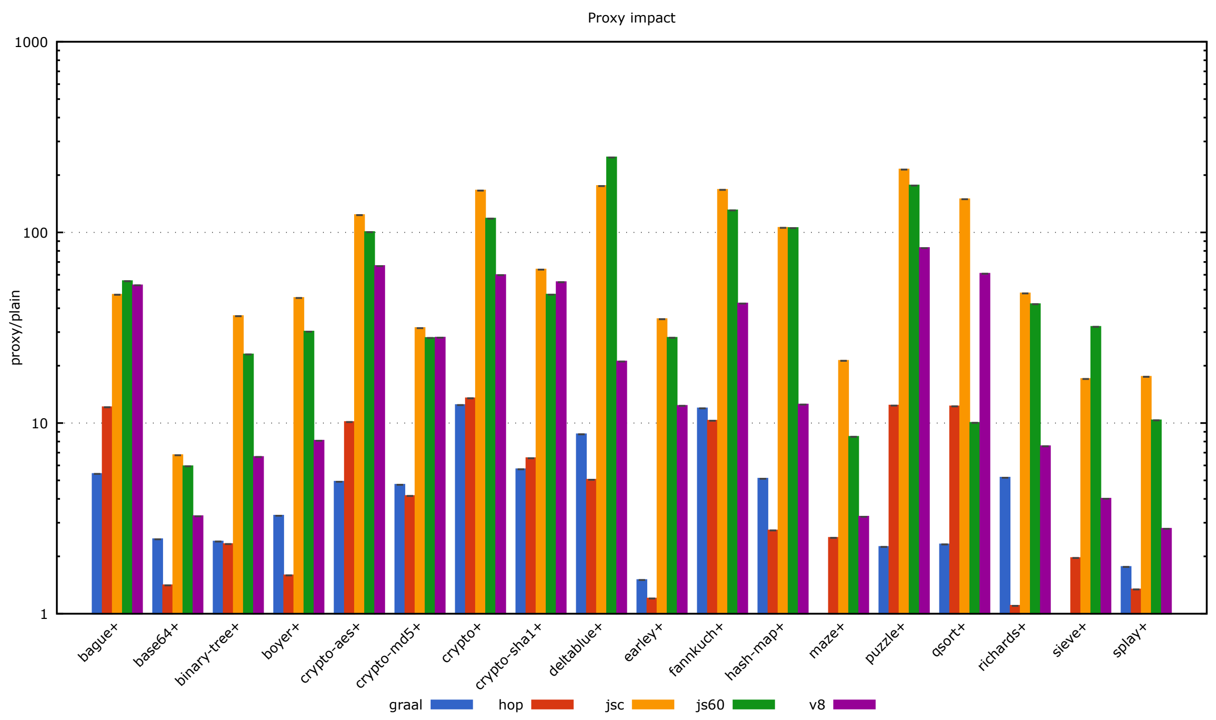

Figure 5. Impact of using proxies on performance. Each bar represents the ratio between the original test and the proxied version. Smaller is better; log scale.

slowly. Using floating point intensive benchmarks would measure these mismatches, and not the proxy access.

5.1 Performance of Proxy Objects

The main objective of the optimizations presented in this paper is to improve proxies performance so that they can be used in more situations. In this section we compare their performance with Hop and with the other JavaScript implementations.

Proxies are relatively new to JavaScript and no standard performance test is available, so we created our own for the experiment. We reused mid-size programs coming from different classical JavaScript benchmark suites (Octane, Sunspider, Jetstream, Shootout, and some other tests used in previous JavaScript evaluation reports) that we modified so that each object or array creation is replaced with an equivalent proxy creation that simply executes the corresponding action to its target object.

These modifications applied to the test correspond to the worse-case situation, as all objects and arrays are trapped by proxies. This over-emphasizes the impact of proxy objects, as needed for this experiment, and it correspond to a upper bound for slowdown factors due to proxy accesses in real-life programs.

We have calibrated each test so that the slowest system executes within 100 seconds and the Hop execution is as close as possible to 10 seconds. The results are given in Figure 4. They show that Hop outperforms all systems for all tests except binary-tree+ where only Graal is able to get slightly better performance and on sieve+ where V8 is able to perform about 40% faster. For all other tests the performance difference between Hop and other implementations is significant. For instance, on the traditional deltablue+ and crypto-md5+ tests, Hop performs about 5× times faster than all other systems. Notice that Graal is not able to run the maze and sieve tests; in both cases, a stack overflow happens and Graal exits.

To give a more precise comparison of all proxy implementations, we also measure proxy impact on general performance. Figure 5 gives this It shows the execution times of proxied tests divided by the execution times of the original corresponding tests. Notice that this figure uses a logarithmic scale. This test shows that the performance gap between Hop and other systems is even more important that one may deduce from Figure 4. On most tests, Hop performance is generally in between 2× to 4× slower than systems like V8 [Serrano 2018; Serrano and Feeley 2019], but its proxy implementation bridges that gap and even enables it to outperform other systems. For instance, on the Octane earley test, the impact of introducing proxy objects is a slowdown of 1.66 for Hop, but 11.65 for V8, 29.39 for Js60, and 36.57 for Jsc. Only Graal is able to show comparable slowdown but notice that this system overall is generally significantly slower.

Figure 5 also gives a general picture of the performance Hop can deliver for proxy objects. For 11 out of the 16 tests, the proxied programs are within a 10x range of their corresponding original ones. All programs that show pathologically bad performance use arrays extensively and the slowdown seems inevitable as it mostly comes from turning the fast memory reads and writes of the original version into function calls in the proxy versions. Some programs, such as qsort, also suffer from an extra problem. They use large arrays whose indexes overflow the limit from which index properties are stored in the table (as discussed in Section 3.1), which causes extra memory allocations.

We have conducted another experiment that evaluates the impact of proxy objects when not all allocation sites have proxy wrappers. For each test, we measured the execution times when the percentage of proxied objects varies from 100% to 10%. Figure 6 shows the result of that experiment for crypto-aes+. The performance of all systems increases when the percentage of proxies decreases. This confirms the results of Figure 5. We have observed similar results for all tests but for base64+, fannkuch+, hash-map+, and puzzle+. For these four tests, the performance of Hop executions behaves as it does for crypto-aes+ but Jsc, Js60, and V8 have more chaotic executions reflected by discontinuous time curves. We are unsure what causes those irregularities, as we do not understand those systems' runtime support for proxies well enough.

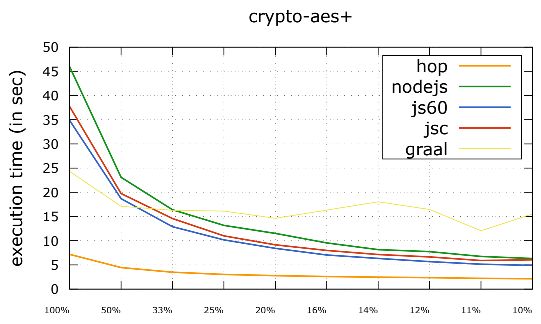

Figure 6. Execution times depending on the percentage of proxied objects. Smaller is better; linear scale.

Figure 7 shows the impact of the proxy optimizations deployed in Hop. It presents the performance of the optimizations used separately. Obviously, the two most influential optimizations are the dynamic property names (**contrib. #2**) and the proxy internal inline caches (**contrib. #3c**). Disabling internal inline caches always slows down execution, by a significant factor (up to almost two with our tests). We then conclude that this optimization should always be applied.

The dynamic property names optimization has a different impact. It it slightly slows down some tests and it accelerates significantly others. We have measured the cache misses ratio of all these tests and we have observed that all the tests that suffer slowdowns are array intensive. Hop does not use inline caches for accessing arrays with integer indexes because it relies as much as possible on faster indexed memory accesses. As a consequence, for array accesses the test, `obj->hclass == name->readcache->hclass` (see Section 3.2) always fails. The performance penalty observed in Figure 7 comes exclusively from this test. For all other tests, that is for all tests that spend a significant portion time on object access, the dynamic property name is the most beneficial optimization, even for tests, such as richards+, for which some name overloading causes dynamic inline cache misses. This experiment also shows that except for one benchmark (crypto+), allocating caches per allocation-site instead of per-allocated proxy (contrib. #3d) is highly beneficial.

5.2 Impact of String Caches

The techniques presented in this paper have very little impact on the rest of the implementation of the runtime system. The only modification that hurts performance is the need for extra fields in the string value implementation. To measure that impact, we compare the performance of two versions of Hop that differ only by their representation of strings and the dynamic property optimization described in Section 3. We have selected tests that do not use dynamic property names so the lack of dynamic optimization imposes no penalty. The result are presented in Figure 8.

The differences between execution times is in the range ±3% and even close to 0% for many programs. We have observed only three tests for which it yields to a memory footprint increase. As only strings used as property names need to be extended with inline caches, it might be that a clever string encoding could eliminate the allocation of that memory slot for non-name strings, similarly to what we did when merging the right part of ropes and the write inline caches.

5.3 Performance of Dynamic Property Names

In order to improve our understanding of the good Hop proxy performance we designed various micro-benchmarks that attempt to isolate the speed of the dynamic property name access used by proxy objects.

The first test consists of a simple loop that accesses all of the properties an object holds. The access to object properties is implemented using two variants. The dynamic version is implemented as follows:

```
for( let i=keys.length-1; i>=0; i-- ) { r+=a[keys[i]]; }
```

In the static version the expression `a[keys[i]]` is replaced with a `switch` that branches according to all possible names used in the program. The results are presented in Figure 9.

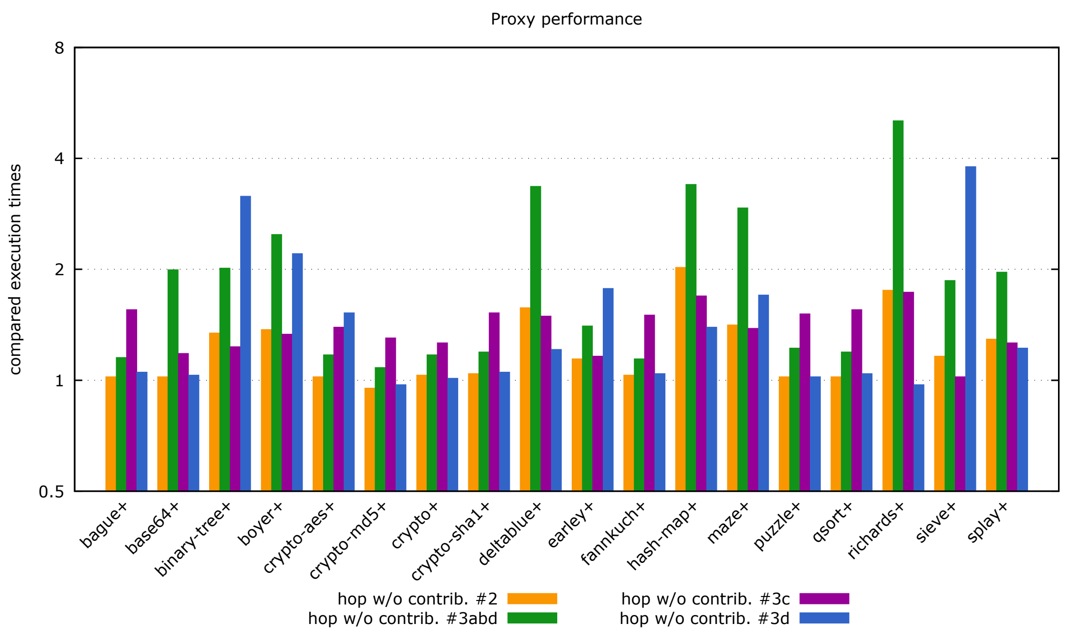

Figure 7. Impact of each proxy optimization compared to the baseline compiler (all optimizations switched on). Smaller is better; log scale.

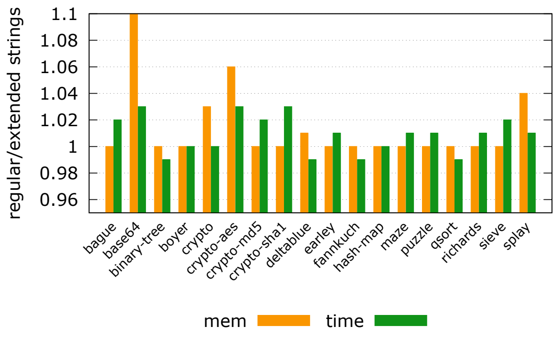

Figure 8. Evaluation of the cost of extra information in strings on non-proxied programs. For each tests, the "time" bar shows the execution time with normal strings divided by the time of execution time with extended strings. The "mem" bar shows the memory consumption comparison. The closer to 1, the smaller the impact.

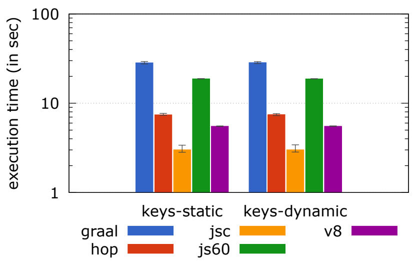

Figure 9. Comparing the raw performance of JavaScript implementations for static and dynamic property names. Smaller is better; log scale.

With the exception of Hop, all the implementations impose a significant cost for the dynamic version. The good performance for both Hop versions establishes the benefit of the dynamic property accesses optimization presented in Section 3.2.

Next, we measured the performance of index property names. We know, from examining the source code, that V8 and Js60 (like Hop) both use string representations that hold references to the corresponding index when the string's content is a number. That said, we still see a performance gap between them and Hop. These experiments report on the gap, but we do not have an explanation for the gap based on the representations that V8 and SpiderMonkey are using. While we cannot claim to fully understand the complete details of other engines' representations, the picture that these performance results paint suggests that Hop's representations are the right choices.

The test foridx measures the performance of array indexes used as integers. It repeats the loop `r=0; for(let i in a) r+=(+i)` where `a` is an array and `r` an integer. The test forin measures array performance of string array indexes with the loop `for(let i in a) r+=a[i]`. The test forkey is similar to forin but array indexes are first stored in a separated array and the loop is `for(let i=0; i<l; i++) r+=a[k[i]]`. The test forstr is similar to foridx, but indexes are concatenated as

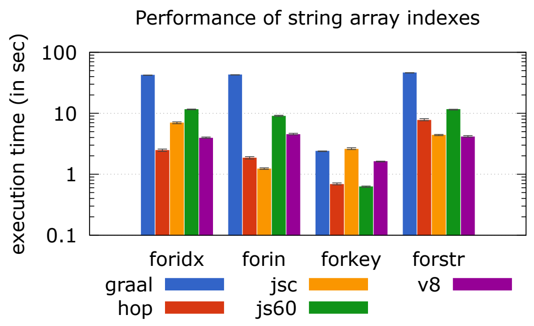

Figure 10. Comparing the raw performance of JavaScript implementations for array string array accesses. Smaller is better; log scale.

strings. The loop is `s=""; for(let i in a) s+=i`. Figure 10 shows the scores for these tests. It reveals that systems come with some trade offs. Jsc is fast for accessing arrays in the loop but all the regular string operations on indexes are slow. V8 is reasonably fast for using array indexes as integers but relatively so for all other options. Js60 is very fast for accessing array with pre-computed string indexes but slower for all other tests. We think that the test shows the general good performance of our contribution #1 presented in Section 3.1. It is the fastest for the numerical conversion (foridx) and consistently second for forin and forkey. It is slightly slower for forstr but for another reason. This test keeps allocating small strings that have a very short life time. This pattern penalizes the non-copying collector used by Hop.

6 Related Work

The essential part of the literature regarding proxies focuses on their design [Van Cutsem and Miller 2010, 2013]. Very little is said about their performance. A blog post from 2016 mentions that poor performance might jeopardize their adoption [Karpov 2016]. Another post from Google [Lekova 2017] acknowledges the performance problem and points out the increase of popularity of proxy objects. It presents improvements applied from Node v8.4.0 and v9.0.0 and shows that the performance has improved by about 50% from the first version to the second. The optimizations they describe mostly focus on the implementation of the construction of proxies themselves but they do not address the problem of efficiently implementing dynamic property access nor the efficient integration of the proxy accesses into the inline caching machinery. For our experiments, we have used V8 embedded in Node v10.15.2. It is a much more recent version than the ones described in the blog post and we have shown that the techniques we present here enables Hop to be more than twice as fast as Node when proxies are involved, despite Node having a better baseline performance.

Bauman et al. [2017] present an optimization of Racket's *chaperones* [Strickland et al. 2012b] (a feature of Racket that is similar in spirit to JavaScript proxy objects, but predates them) to improve the performance of gradual typing. While Racket's chaperones and JavaScript proxies are similar in spirit, the semantic details differ, and the details required for good performance do not carry over from one system to another. More precisely, Racket's impersonators are read-only and Racket's structures (Racket's structures correspond to JavaScript objects) have a statically-known structure, completely avoiding the hassle of an efficient implementation of the prototype chain lookup. Additionally, the relevant main contribution of Bauman et al. [2017]'s work consists of adapting the techniques of hidden classes, something that JavaScript implementations take for granted and is insufficient to deliver decent performance for proxies.

Safe TypeScript is a sound variation of TypeScript [Microsoft 2013]. Rastogi et al. [2015] report an overhead of 2.4-72× slowdown with respect to unsound TypeScript. Type safe safety is implemented using JavaScript proxies and V8 is the platform they used for their experiment. The optimizations we present here should have a significant benefit to their system.

7 Conclusion

Efficient proxies are critical for unlocking a host of important services that programming languages and their libraries provide. The one attracting the most ink in the programming languages literature is clearly gradual typing [Greenman et al. 2019], but the blogosphere is alive with interesting uses, including debugging async functions [Gimeno 2018], implicit defaults for subtypes [Barrasso 2019], debugging imperative code (when the buggy modification is long gone from the stack) [Rauschmeyer 2018], revocation of resources [Opia 2018] and more.

This paper takes a first step towards efficient proxies in JavaScript, describing some key optimizations and showing that the performance can be significantly improved at negligible cost. Better yet, adding these optimizations to Hop was straightforward (once we uncovered their value) and is likely to be straightforward in any other performant JavaScript implementation.

The performance boost that these optimizations give is significant, too. Simply adding proxies to the standard Octane benchmark suite shows a 100× slowdown (or sometimes even more) for modern implementations. The optimizations we propose in this paper allow Hop to reduce the slowdown to 10× or even less in many tests. It enables Hop to generally take over other JavaScript implementations, despite having a weaker baseline performance.

References

R. Artoul. 2015. Javascript Hidden Classes and Inline Caching in V8. http://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html.

T. Barrasso. 2019. A practical guide to Javascript Proxy. https://blog.bitsrc.io/a-practical-guide-to-es6-proxy-229079c3c2f0.

S. Bauman, C-F. Bolz-Tereick, J. Siek, and S. Tobin-Hochstadt. 2017. Sound Gradual Typing: Only Mostly Dead. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 54 (Oct. 2017), 24 pages. https://doi.org/10.1145/3133878

H.J. Boehm and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience* 18, 9 (Sept. 1988), 807–820.

H-J. Boehm, R. Atkinson, and M. Plass. 1995. Ropes: an Alternative to Strings. *Software: Practice and Experience* 25, 12 (Dec. 1995), 1315–1331.

C. Bruni. 2017. Fast Properties in V8. https://v8project.blogspot.fr/2017/08/fast-properties.html.

C. Chambers and D. Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *Conference Proceedings on Programming Language Design and Implementation (PLDI '89)*. ACM, USA, 146–161.

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. ACM, USA, 49–70.

P. Deutsch and A. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, USA, 297–302.

ECMA International. 2011. *Standard ECMA-262 - ECMAScript Language Specification* (5.1 ed.). http://www.ecma-international.org/publications/standards/Ecma-262.htm

ECMA International. 2015. *Standard ECMA-262 - ECMAScript Language Specification* (6.0 ed.). http://www.ecma-international.org/ecma-262/6.0/

A. Gimeno. 2018. How to use JavaScript Proxies for Fun and Profit. https://medium.com/dailyjs/how-to-use-javascript-proxies-for-fun-and-profit-365579d4a9f8.

Google. 2018. V8 JavaScript Engine. http://developers.google.com/v8.

Google. 2019. name.h. https://github.com/v8/v8/blob/4b9b23521e6fd42373ebbcb20ebe03bf445494f9/src/objects/name.h#L99-L104.

B. Greenman, A. Takikawa, M. New, D. Feltey, R. Findler, J. Vitek, and M. Felleisen. 2019. How to Evaluate the Performance of Gradual Type Systems. *Journal of Functional Programming* 29 (2019).

U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. 21–38.

V. Karpov. 2016. Thoughts on ES6 Proxies Performance. http://thecodebarbarian.com/thoughts-on-es6-proxies-performance.

M. Keil, S. Guria, A. Schlegel, M. Gefken, and P. Thiemann. 2015. Transparent Object Proxies in JavaScript. In *29th European Conference on Object-Oriented Programming (ECOOP 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Tang Boyland (Ed.), Vol. 37. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 149–173. https://doi.org/10.4230/LIPIcs.ECOOP.2015.149

R. Kelsey, W. Clinger, and J. Rees. 1998. The Revised(5) Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (Sept. 1998).

M. Lekova. 2017. Optimizing ES2015 proxies in V8. https://v8.dev/blog/optimizing-proxies.

Microsoft. 2013. TypeSscript, Language Specification, version 0.9.5.

Mozilla. 2019a. Proxy. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy.

Mozilla. 2019b. Strings. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String.

Mozilla-central. 2019. StringType.h. https://searchfox.org/mozilla-central/rev/7ed8e2d3d1d7a1464ba42763a33fd2e60efcaedc/js/src/vm/StringType.h#422-428.

C. Opia. 2018. A quick intro to JavaScript Proxies. https://www.freecodecamp.org/news/a-quick-intro-to-javascript-proxies-55695ddc4f98/.

Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. 2015. Safe &#38; Efficient Gradual Typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 167–180. https://doi.org/10.1145/2676726.2676971

A. Rauschmeyer. 2018. *Exploring ES6*.

B. Serpette and M. Serrano. 2002. Compiling Scheme to JVM bytecode: a performance study. In *7th* ACM SIGPLAN *Int'l Conference on Functional Programming (ICFP)*. (taux d'acceptation : 24/76), Pittsburgh, Pensylvanie, USA.

M. Serrano. 2018. JavaScript AOT Compilation. In *14th Dynamic Language Symposium (DLS)*. Boston, USA. https://doi.org/10.1145/3276945.3276950

M. Serrano and M. Feeley. 2019. Property Caches Revisited. In *Proceedings of the 28th Compiler Construction Conference (CC'19)*. Washington, USA. https://doi.org/10.1145/3302516.3307344

S. Strickland, R. Findler, M. Flatt, and S. Tobin-Hocshstard. 2012a. Chaperones and impersonators: Run-time support for reasonable interposition. *ACM SIGPLAN Notices* 47 (10 2012). https://doi.org/10.1145/2384616.2384685

S. Strickland, S. Tobin-hochstadt, R. Findler, and M. Flatt. 2012b. M.: Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. Arizona, USA, 943–962.

S. Thompson. 2015. Design Elements. https://github.com/v8/v8/wiki/Design%20Elements.

T. Van Cutsem and M. Miller. 2010. Proxies: Design Principles for Robust Object-oriented Intercession APIs. *Proceedings of the 6th Symposium on Dynamic Languages, DLS '10* 45, 59–72. https://doi.org/10.1145/1869631.1869638

T. Van Cutsem and M. Miller. 2013. Trustworthy Proxies - Virtualizing Objects with Invariants. In *27th European Conference on Object-Oriented Programming (ECOOP 2013)*.

Wikipedia. 2019. Rope (data structure). https://en.wikipedia.org/wiki/Rope_%28data_structure%29.