Property Caches Revisited

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

Marc Feeley
Université de Montréal
Montréal, Québec, Canada
feeley@iro.umontreal.ca

ABSTRACT

Property caches are a well-known technique invented over 30 years ago to improve dynamic object accesses. They have been adapted to JavaScript, which they have greatly contributed to accelerate. However, this technique is applicable only when some constraints are satisfied by the objects, the properties, and the property access sites. In this work, we propose enhancements to improve two common usage patterns: *prototype accesses* and *megamorphic accesses*. We have implemented these in the Hopc AOT JavaScript compiler and we have measured their impact. We observe that they effectively complement traditional caches. They reduce cache misses and consequently accelerate execution. Moreover, they do not cause a slowdown in the handling of the other usage patterns.

CCS CONCEPTS

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

KEYWORDS

JavaScript, Compilation, AOT, Hidden Classes

ACM Reference Format:
Manuel Serrano and Marc Feeley. 2019. Property Caches Revisited. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3302516.3307344

1 INTRODUCTION

JavaScript objects are dynamic. At any moment of their lifetime, properties can be added or deleted. In principle a property access requires a lookup in the object itself, and, possibly, in all the objects forming its prototype chain [ECMA International 2011, 2015]. All fast JavaScript implementations deploy strategies to implement this lookup operation in nearly constant time. They generally rely on two ingredients: *hidden classes* and *property caches*. Hidden classes describe object memory layouts. Property caches use these descriptions to access objects directly, avoiding the normal name lookup operations. Hidden classes and property caches make property accesses comparable in speed to field accesses of traditional languages like C and Java.

Hidden classes and property caches are not new. They were invented for Self, the first dynamically typed prototype-based languages [Chambers and Ungar 1989; Chambers et al. 1989], following Smalltalk's idea that already used caches at that time for optimizing method calls [Deutsch and Schiffman 1984]. For the past ten years they have enjoyed a revival of interest after it was shown how effective they are at improving Object-Oriented languages performance in general [Wößand *et al.* 2014] and specially JavaScript [Google 2018]. Today most JavaScript implementations use them [Ahn et al. 2014; Gong et al. 2015; Schneider 2012]. Hidden classes and property caches apply in specific situations, which unfortunately means that some accesses are unoptimized or not treated very efficiently.

1. **Prototype properties problem**: hidden classes and property caches optimize accesses of properties directly stored in the object. They do not optimize accesses of properties stored in one of the objects composing the prototype chain.
2. **Polymorphic properties problem**, as property caches require strict hidden class equivalence for optimizing accesses, polymorphic data structures and polymorphic method invocations need special treatment to not be left unoptimized. This has been addressed by the *Polymorphic Inline Cache* technique proposed by Holzle *et al.* [Hölzle et al. 1991], which resorts to a dynamic search in the cache history. As a linear or binary search is involved, it is not as efficient as plain property caches.

We propose solutions to these problems that might become important with the advent of ECMAScript 6 class-like construct that simplifies the programming of polymorphic patterns [ECMA International 2015]. At the cost of one extra test inserted at each property access, we optimize prototype property accesses. Trading memory space for speed, we propose *cache property tables* that enable accessing polymorphic objects in constant time. For the analogy with C++ virtual tables we call these cache tables *vtables*.

The paper is organized as follows. In Section 2 we briefly present hidden classes and property caches. This is an introduction for readers unfamiliar with these implementation techniques. In Section 3 we dig deeper in the specificities of JavaScript property reads and we present our technique for optimizing prototype accesses, which also applies to improve getter accesses. In Section 4 we show that hidden classes fail at optimizing polymorphic objects and we introduce the *vtables* as a means to eliminate this problem. The presented techniques and optimizations have been implemented in Hopc [Serrano 2018], an AOT JavaScript compiler that targets server-side computations. Hopc compiles modules separately and the techniques we propose are compatible with this implementation schema. We outline their implementation in Section 5 and we show their effectiveness using an experimental evaluation presented in Section 6. Finally, we present related work in Section 7.

CC '19, February 16–17, 2019, Washington, DC, USA

ACM ISBN 978-1-4503-6277-1/19/02…$15.00
https://doi.org/10.1145/3302516.3307344

2 HIDDEN CLASSES AND PROPERTY CACHES

According to the JavaScript specification [ECMA International 2015], accessing an object property involves the following steps:

(1) convert the property name into a string *S*;
(2) if the object owns a property *S*, return its value;
(3) if the object has a prototype, restart at step (2) with the prototype object, return `undefined` otherwise.

Both `obj.prop` and `obj["prop"]` access the property with name `"prop"`. The central point of the property access process is step (2). Since new properties can be added or removed at any moment, checking if an object owns a property implies looking for a key (the property name) in a dictionary (the object). Implemented literally, this protocol is orders of magnitude slower than those of languages for which reading a structure field is a single memory read whose address is computed by adding to a base pointer an offset known at compile time.

The classical method for optimizing property accesses consists in associating with each object a *hidden class* and with each access a *lookup cache* [Chambers and Ungar 1989; Chambers et al. 1989; Deutsch and Schiffman 1984]. When the property name is statically known, a very frequent case, a property access `obj.prop` can be implemented as follows, using C as the implementation language:

```
if( obj->hclass == cache.hclass ) {
   val = obj->elements[ cache.index ]; // cache hit
} else {
   val = cacheReadMiss( obj, &cache ); // cache miss
}
```

On a cache miss, the `cache`'s `hclass` attribute is updated with the object's hidden class. The object hidden class is updated each time a property is added or removed so the condition `obj->hclass == cache->hclass` only holds for objects that have exactly the same structure. Figure 1 shows the memory layout of an object `obj` owning three properties `x`, `y`, and `z`, its associated hidden class, and a possible property cache after a cache miss.

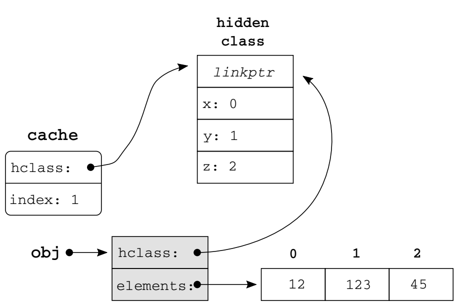

Figure 1: A property cache after a miss on property "y".

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 (`obj->hclass == cache.hclass`). If self modifying code is used, as JIT compilers do, the two memory reads from `cache` can be eliminated because the `hclass` and the `index` values can be directly stored in the machine instructions that perform the comparison and the fetch operation. In that case, the property cache is called an *inline cache* [Chambers et al. 1989]. It has been observed that inline caches improve the performance over plain property caches up to 25% [Chambers et al. 1989] on computers of the time[1].

A hidden class characterizes an object memory layout at a certain moment of its lifetime. A simple way to associate hidden classes with objects is to construct dynamically a deterministic automaton whose transitions are labeled with added or deleted property names and states represent hidden classes. Several documents [Artoul 2015; Bruni 2017; Deutsch and Schiffman 1984; Google 2018; Thompson 2015] explain how hidden classes are built and how inline caches work in actual industrial implementations.

```
function readX( obj ) {
   return obj.x;
}

function test( count, N ) {
   let os = [{ x: 21, y: 31 }, { x: 12, y: 123, z: 45 }];
   let s = 0;
   for( let i = 0; i < count; i++ ) {
      let o = os[ i % N ];
      s = readX( o );
   }
   return s;
}
```

Figure 2: Monomorphic accesses (`N=1`) vs polymorphic accesses (`N=2`).

Let us consider the example of Figure 2 that exercices a property access in a loop. When the parameter `N` is `1` the `obj.x` property access in `readX` is always performed on the same object. When `N` is `2`, the object and its type change at each iteration of the loop and access `obj.x` is said to be *polymorphic*. The simple property cache described previously always misses in this case.

For accelerating polymorphic accesses Hölzle *et al.* [Hölzle et al. 1991] have proposed *polymorphic inline caches* (PICs) which extend cache sizes. Instead of recording only the last hidden class, multiple or even all classes are stored in the cache and are probed in sequence. JIT compilers can generate these tests on demand. For an AOT compiler they can be implemented using a loop as:

```
for( i = 0; i < cache.size; i++ ) {
   if( obj->hclass == cache.hclasses[ i ] ) {
      val = obj->elements[ cache.indices[ i ] ]; // cache hit
      goto __done;
   }
}
val = cacheReadMiss( obj, &cache ); // cache miss
__done:
```

The study reports that PICs greatly accelerate some programs (for instance, by 47% for the `richards` benchmark). PICs are nowadays deployed in modern JavaScript implementations (see Section 6.2). However, as acknowledged by the authors, PICs are inappropriate when the degree of polymorphism increases greatly, in which case the access is said to be *megamorphic*. For that, they

[1]We are not aware of any study that would have updated this result for contemporary architectures. We think such a study would be of high interest for the community as in our experience, and despite all our efforts, we have not been able to obtain any measurable speedup by turning caches into inline caches on modern architectures.

suggest using a binary search instead. In Section 4 we propose an alternative approach inspired by virtual tables and in Section 6 we compare its performance against classical PICs. We show that it performs faster without incurring a code size or a memory footprint increase.

3 ACCESSOR AND PROTOTYPE PROPERTIES

Section 2 sketches the main ideas and principles that govern hidden classes and inline caches. This section focuses on the difficulties JavaScript specificities raise for implementing fast accesses, namely *accessor properties* and *prototype chain accesses* and the solutions we propose to mitigate these problems.

3.1 Accessor Properties

JavaScript supports two kinds of properties: *value properties* and *accessor properties.* Value properties are stored in the objects that own them. Accessor properties are pairs of functions invoked when properties are read or written (*aka.* getters and setters). The property cache schema presented in Section 2 does not cope with accessor properties and JavaScript implementations that only rely on that method do not optimize them (see Section 6.2). To improve the performance of property accesses, Hopc extends the information stored in the property cache and adds one extra test after a cache miss.

```
1 if( obj->hclass == cache.hclass ) {
2    val = obj->elements[ cache.index ];
3 } else if( obj->hclass == cache.aclass ) {
4    // test the "accessor class" for an accessor cache hit
5    val = obj->elements[ cache.index ]( obj );
6 } else {
7    val = cacheReadMiss( obj, &cache );
8 }
```

On a cache miss, if the property is found in the object, two cases are now considered.

- If it is a *value property*, the `hclass` field of the property cache is filled as before. A subsequent test at line 1 will then succeed for an object of that type.
- If the property is an *accessor property*, the extra `aclass` field of the cache is filled with the object’s hidden class, the cache’s index field is set to the index of the getter and the `hclass` field is reset. The test at line 3 will then succeed for an object of that type.

Figure 3 shows the property cache after a cache miss on property `x`, assuming that the example of Figure 2 has been modified and the objects are now defined as:

```
let os = [ { get x() { return 21 }, y: 31 },
           { get x() { return 12 }, y: 123 } ];
```

Enhanced property caches enable accessor properties to be handled almost as efficiently as regular properties. There is only an overhead of one extra test for the former. Figure 12 (Section 6.2) presents the performance evaluation of the modified micro benchmark.

3.2 Prototype Chain Accesses

Hidden classes and property caches cannot efficiently handle properties located in prototype objects as they compare the class of

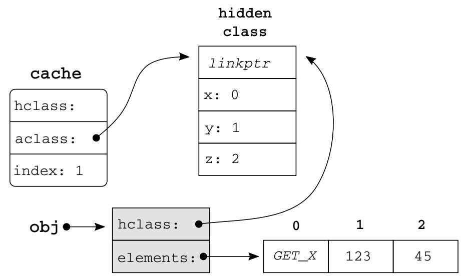

Figure 3: A property cache after a miss on accessor property “x”.

the object from which the property is fetched, which is not the object that owns the property when it is stored in an object of the prototype chain. To optimize these accesses as well, Hopc extends the property caches with two additions: the class of the object of the prototype chain that actually owns the property and the owner itself. The cache probe is modified as follows:

```
if( obj->hclass == cache.hclass ) {
   val = obj->elements[ cache.index ];
} else if( obj->hclass == cache.aclass ) {
   val = obj->elements[ cache.index ]( obj );
} else if( obj->hclass == cache.pclass ) {
   // test the "prototype class" for a prototype cache hit; the property
   // is found in the prototype chain
   val = cache->owner->elements[ cache.index ];
} else {
   val = cacheReadMiss( obj, &cache );
}
```

On a cache miss, a full lookup in the whole prototype chain is executed. If the property is directly found in the object, the cache is filled as previously described. If the value found is not an accessor value, the object owning the property, *i.e.,* the first object in the prototype chain that defines the searched property, is stored into the property cache (the `owner` field). Let us illustrate these new caches with the following objects:

```
let p = { cnt: 12345 };
let o = { __proto__: p, x: 12, y: 31 };
```

The cache configuration after filling the cache when accessing `o.cnt` is presented in Figure 4.

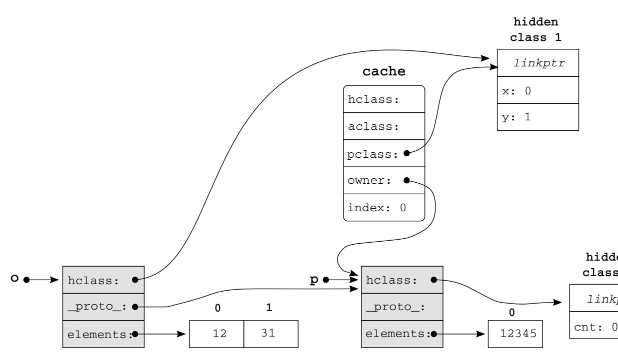

Figure 4: A prototype cache after a miss on prototype property “cnt”.

Handling prototypes efficiently in the property cache requires modifying the hidden class construction. With this modification, two objects share the same hidden class if *i)* they have the same memory layout, and *ii)* their prototype object is the same. For this, the `__proto__` property is handled differently than other properties. The link between two hidden classes is labeled with property names, except when that property is the `__proto__` property in which case the transition is labeled with the actual prototype object. Let us consider the following objects:

```
let p1 = { cnt: 45 };
let p2 = { cnt: 0,
           toString: function() {return this.x+","+this.y} };
let os = [{ x: 1, y: 2, __proto__: p1 },
          { x: 4, y: 5, __proto__: p2 }];
```

Four hidden classes are needed for objects `os[0]` and `os[1]`, see Figure 5. The transition from hidden classes 0, 1, and 2 are labeled with the property names `x` and `y`. The transitions from the class 2 to class 3 and class 4 are labeled with the two JavaScript objects `p1` and `p2` that are respectively the prototype of the objects `os[0]` and `os[1]`.

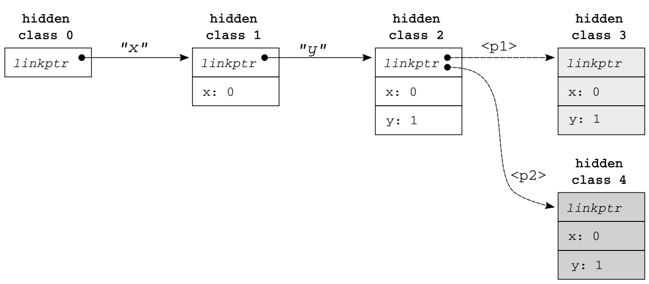

Figure 5: Hidden classes with prototype objects valued transitions. Hidden classes 2, 3, and 4, are all distinguished heap allocated objects.

Using prototype objects in the hidden class tree hierarchy and storing the prototype objects in the property caches requires the preservation of two runtime invariants.

(1) Objects' hidden class must be in sync with objects' prototype during their whole lifetimes. That is, prototype objects used to label transitions between two hidden classes must correspond to objects' prototype objects.
(2) Indexes of property cache prototype must be in sync with the actual structure of prototype objects.

Let us consider the following example that illustrates invariant 1.

```
1 function readCNT( obj ) { return obj.cnt }
2 let p1 = { cnt: 45 }, p2 = { cnt: 63 };
3 let o = { __proto__: p1 };
4
5 readCNT( o );
6 o.__proto__ = p2;
7 readCNT( o );
```

The call on line 5 returns `45` while the call on line 7 returns `63`. This is because `o`'s prototype has changed between the two calls. The preservation of invariant 1 forces the runtime to invalidate the cache used in `readCNT` when the `__proto__` property is updated. Invariant 2 is illustrated by the following example:

```
1 let p2 = { cnt: 23 }, p1 = { cnt: 45, __proto__: p2 };
2 let o = { __proto__: p1 };
3
4 readCNT( o );
5 delete p1.cnt;
6 readCNT( o );
```

The call on line 4 returns 45 while the call on line 6 returns 23. This is because on line 4, the property `cnt` is found in `p1` but on line 6 it is found in `p2`, as the property has been removed from `p1` on line 5. Here again, the modification of `p1` invalidates the cache used in `readCNT`.

In Hopc the two invariants are enforced with the following measures:

(1) all modifications of any prototype property (detected by the use of property name `__proto__` or by the use of the function `setPrototypeOf`) invalidate *all* the property caches `pclass` fields and it creates a fresh copy of the object's hidden class, distinguished from all already existing hidden classes;
(2) any deletion of an object property invalidates all the property cache `pclass` fields;
(3) any overriding of a property already defined in an object's prototype chain, invalidates all cache `pclass` fields. This is detected on a write cache misses, without the need of any additional object fields.

These three measures require invalidating all property cache `pclass` fields. This might be an expensive operation as it depends on the size of the program (the more read and write locations in the source, the more expensive the invalidation is). However, we have observed that in practice these invalidations are rare. Intuitively this is because prototype objects are mainly set during the initialization phase of an application, and because property overriding and property deletion are rare. To go beyond that intuition, we have measured how frequent these operations are on a set of JavaScript programs. The results of this evaluation are presented in Section 6, Figure 14.

4 VIRTUAL TABLES

Simple property caches assume monomorphic programs. They effectively optimize repetitive accesses to objects that share the exact same structure denoted by their identical hidden classes. Polymorphic property caches have been proposed to optimize polymorphic programs. They are effective as long as the degree of polymorphism is small, which might no longer hold in JavaScript with the recent adoption of a class-based object-oriented style. Former JavaScript versions were fostering prototype-based programming but ECMAScript 2015 added class declarations to the language, which, although elaborated over object prototypes, make it easier to implement the classical programming patterns of traditional class-based object-oriented languages. This will naturally favor polymorphic programming, and might demand polymorphic and megamorphic property accesses to be handled more efficiently. Let us consider the following JavaScript class declarations:

```
class Point {
   constructor( x ) { this.x = x; }
   readX() { return this.x }
}
class Point2D extends Point {
```

```
6    constructor( x, y ) { super( x ); this.y = y; }
7  }
8  class Point3D extends Point2D {
9    constructor( x, y, z ) { super( x, y ); this.z = z; }
10 }
```

Instances of the three classes `Point`, `Point2D`, and `Point3D` will be associated with different hidden classes and accessing the property `x` in the method `readX` (line 3) will be polymorphic. As already mentioned in Section 2 such situations can be handled by encoding several tests when probing a cache. The efficiency depends on the number of different hidden classes. We propose an alternative technique that handles polymorphic property accesses more efficiently and whose complexity is independent of the degree of polymorphism. By analogy with C++, we reuse the terminology *virtual tables* (henceforth vtables) for denoting its main ingredient.

A polymorphic access is characterized by objects belonging to different classes, whether these objects are in an inheritance-like relationship or not. As hidden classes are of no help in these situations, we second them with dynamic data structures that keep track of the accesses that are observed at a particular property access point in the program.

Hidden classes are extended with virtual tables and property caches with virtual index fields (`hclass->vtable` and `cache.vindex`). Monomorphic accesses are handled as before: on a cache hit, the comparison with `obj->hclass` or `obj->pclass` succeeds. On a cache miss the vtable is checked before calling the slow `cacheReadMiss` routine. The read property sequence is:

```
if( obj->hclass == cache.hclass ) {
  val = obj->elements[cache.index];
} else if( obj->hclass == cache.aclass ) {
  ... /* as before */ ...
} else {
  if( obj->hclass->vtable[cache.vindex] >= 0 ) {
    val = obj->elements[obj->hclass->vtable[cache.vindex]];
  } else {
    val = cacheReadMiss( obj, &cache );
  }
}
```

When cache misses are observed on a particular property access, a new vtable is allocated and stored in the hidden class. Its size is given by the value of `vindex`. The vtable records that for that particular property access, the property is stored at a known index. Let us consider the following program fragment:

```
let os = [{ x: 12345, y: 8 }, { y: -1, x: 543, z: 22 }];
...
   var a53 = o.x;  /* access point #53 */
   ...
   var a86 = o.x;  /* access point #86 */
```

and let us assume that `os[0]` and `os[1]` both flow as `o` at accesses #53 and #86. Until a statically configured cache miss threshold is exhausted the cache configuration for the two access points will oscillate between the two objects. This is depicted in Figure 6. When that threshold is passed, vtables are created for the two hidden classes representing `os[0]` and `os[1]`. They are shown in Figure 7. In this configuration when `os[0]` flows to access point 53, its `hclass` matches neither the cache `hclass`, `aclass`, or `pclass` fields. The vtable of its hidden class is then used. The vtable entry 53, which identifies the first access location in the source code, contains the value 0, which is the index of property `x` in `os[0]`. Object `os[0]`'s hidden class vtable has enabled accessing property `x` without any lookup. The `os[1]`'s vtable will do the same for the other object but note that if only two objects reach points #53 and #86 only one will use a vtable. The other one, more precisely, the last one that raised a cache miss, will enjoy a fast inline cache with a direct hit as vtable hits do not invalidate the current cache configuration.

In summary, in this section we have presented a new mechanism for handling JavaScript polymorphic accesses. For the analogy with C++ we call this mechanism *JavaScript virtual tables*. They take over hidden class comparison on polymorphic reads and writes. Virtual tables are adapted to the object-oriented style fostered by the introduction of classes declaration in ECMAScript 2015. Virtual tables are currently deployed in Hopc and their performance evaluation in this AOT compilation context is presented in Section 6. As they do not rely on any static analysis and as they are created on-demand they could also be easily accommodated by JIT compilers.

5 IMPLEMENTATION

The cache techniques presented in this paper have been designed for Hopc, whose implementation stems directly from the cache techniques described in Sections 2-4, with yet another addition. At compile time Hopc estimates object allocation sizes using techniques inspired by [Clifford 2015]. When a property is added, if it fits the object allocation, it is stored *inline*, which saves one memory access on read and write accesses (see Figure 9). This requires one extra entry in each cache for testing inline properties but as caches are allocated statically (one cache per access in the program), this has no impact on the programs memory consumption. Initially the `elements` field points to the `inline_elements`. The complete property read is presented in Figure 10. The sequence also gives an opportunity to delay the use of vtables for polymorphic accesses. Two distinct hidden classes can be recognized for each access point without using vtables: the first one that matches the `imap` field and the second one that matches the `cmap` field.

Long test sequences have a limited impact on performance as the experiments we have conducted show that a majority of accesses are resolved by the first `iclass` test, a sort of analog to the hardware *level 1* cache hit (Section 6, Figure 14). However long test sequences enlarge the generated code unnecessarily. As Hopc is an AOT compiler, it must rely on static analyses to minimize them. For instance, if no property accessor is ever defined in a compilation unit, *i.e.,* a JavaScript module, then the test against the property cache `aclass` can be omitted, or at least not inlined in the generated code. Another possibility is to avoid generating prototype tests (tests against the `pclass` attribute) for properties that are known to be only assigned in constructors. None of these simplifications have been implemented yet in Hopc and they all constitute directions for future studies.

JIT compilers can reuse enhanced caches without suffering the code growth incurred by long cache test sequences. They can simply monitor the caches and switch dynamically from one representation to another when needed. Figure 14 shows the number of multiple caches, that is the number of program points for which cache hits involve at least two different cache maps. The small number of multiple accesses suggests that JIT compilers will only seldom need

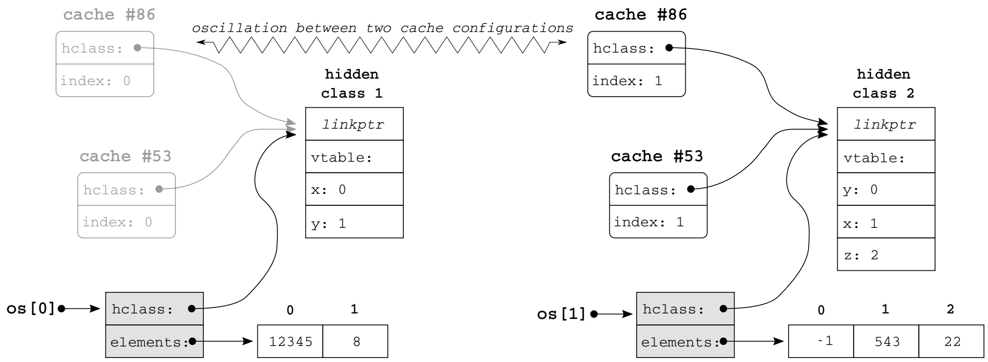

Figure 6: The cache configurations for points #53 and #86 oscillate between `os[0]` and `os[1]` as they both reach the two locations. This causes an expensive cache miss each time the other object reaches one of these two access points.

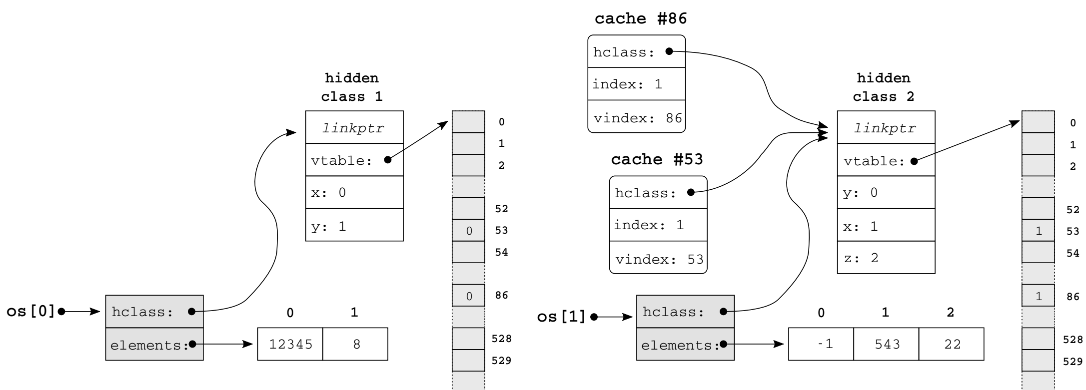

Figure 7: Two virtual tables for `os[0]` and `os[1]` hidden classes, after `o1` and `os[1]` have been accessed at points #53 and #86. For conciseness the cache's `aclass`, `pclass`, and `owner` fields are omitted.

```
total cache hits       : 259x10^6 (99%)   | point property use  |  miss   iclass  hclass   pclass  aclass  vtable
total cache iclass hits: 237x10^6 (91%)   | --------------------+-----------------------------------------------
total cache hclass hits:   4x10^6 (1%)    |  2391        go call |   140        0       0        0       0       0
total cache pclass hits:  18x10^6 (7%)    |  3830      find call |   419        0       0   1x10^6       0       0
total cache aclass hits:        0 (0%)    |  3920    String get  |   141        0       0        0       0       0
total cache vtable hits:        0 (0%)    |  3936    insert call |   280        0       0   1x10^6       0       0
total cache misses     :     8309 (0%)    |  5437 findGrea call  |   140        0       0  11x10^3       0       0
total cache multiple   :  17x10^6 (6%)    |  5532    remove call |   140        0       0  11x10^3       0       0
pmap invalidations     :      298         |  5549       key get  |     1  11x10^3       0        0       0       0
vtables                :      508 (1198b) |  ...
```

Figure 8: Cache profiling. Global statistics on the left, per-site (source character number) report on the right.

to resort to more than one test of the full cache check sequence. So, we conjecture that long test sequences will be harmless in the context of JIT compilers.

In the context of AOT compilation, the optimized solution of JIT compilers can be approached with feedback-directed optimization. This is what we have implemented in Hopc. A first instrumented execution collects the cache information and reports cache use statistics. A second compilation only uses the cache probe that have been used at runtime. Figure 8 shows one such report obtained when executing the `splay.js` benchmark. The left column reports general statistics about caches. It shows that for this execution, 259×10^6 accesses (including gets, puts, and calls) have hit a cache. Among these accesses, 237×10^6, that is 91%, have hit a level 1 cache, that is an `iclass` cache. It also shows, that this particular benchmark has used no accessor map (`aclass`) nor virtual tables (`vtable`). The entry "cache multiple" shows how many caches have been used with at least two different maps (for instance an `iclass` and a `pclass`).

The right part of the report shows cache information per-site, *i.e.,* source character position. For instance, at character number

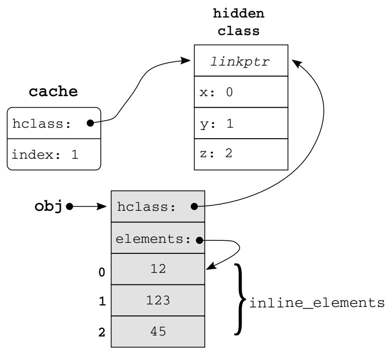

Figure 9: The memory layout of inlined properties. When the properties are allocated *inline* they are stored contiguously to the other object fields.

```
if( obj->hclass == cache.iclass ) {
   val = obj->inline_elements[cache.index];
} else if( obj->hclass == cache.hclass ) {
   val = obj->elements[cache.index];
} else if( obj->hclass == cache.aclass ) {
   val = obj->elements[cache.index]( obj );
} else if( obj->hclass == cache.pclass ) {
   val = cache->owner->elements[cache.index];
} else if( cache.vindex < obj->hclass->vtableLen ) {
   val = obj->elements[obj->hclass->vtable(cache.vindex)];
} else {
   val = cacheReadMiss(obj, &cache);
}
```

Figure 10: The complete Hopc “get” implementation.

5549 the “get” access of property “key” has hit the cache 11×10^3 times and missed it once. These statistics are used to recompile the program and for each access only the logged caches are inlined in the generated code. For instance, for the point 5549, only the `iclass` test is inlined, the other tests are executed in a library function, before raising a cache miss if necessary. The benefit of this approach is studied in Section 6 (Figure 15).

6 EXPERIMENTAL EVALUATION

This section measures the impact of cache techniques we propose, proceeding in two steps. First, it compares Hopc’s performances to those of the other JavaScript implementations, focusing on reading properties that exercise inline caches and hidden classes. For this, micro benchmarks are used. This study shows the benefits our techniques can bring to other systems when simple inline caches fail. Second, it measures the impact of the techniques when compiling standard JavaScript benchmarks. This gives an estimate of the benefits one can expected from applying our techniques to more realistic programs. For that test, only Hopc is used and different compilation modes are compared with one another.

6.1 The Setting

We have applied the same methodology to all our tests. A single machine, an Intel Xeon E5-1650 64-bit running Linux 4.17/Debian, is used. Each test is executed 30 times and the median of wall clock is collected. Unless explicitly specified, the relative standard deviation of each test is less than 5%.

For the system comparison we use Google’s V8 6.2.414.54, JavaScriptCore 4.0 (Jsc), SpiderMonkey C52.3.1 (Js52), and Microsoft’s Chakra 1.10 (Ch). As all these systems use JIT-compilers we have tuned the test to have sufficiently long execution times so that the warm up time of the JIT is negligible.

Benchmarking JavaScript is difficult because of the distance between the language and the current hardware design and because JavaScript optimizations frequently consist in trading memory for speed, which on modern architectures has sometimes unpredictable effects on caches and speculation. Our first assignment has been to design a test program that minimizes the impact of other optimizations the systems deploy. Our starting point was the program given in Figure 2 that we modified so that the `os` variable contains 16 different objects. In that source code, the argument `count` is used to calibrate the benchmark duration and for ensuring that for all benchmarks and systems the fastest execution lasts at least 2 seconds. We have used that same program for all tests. Only the definition of these 16 objects varies from one test to another.

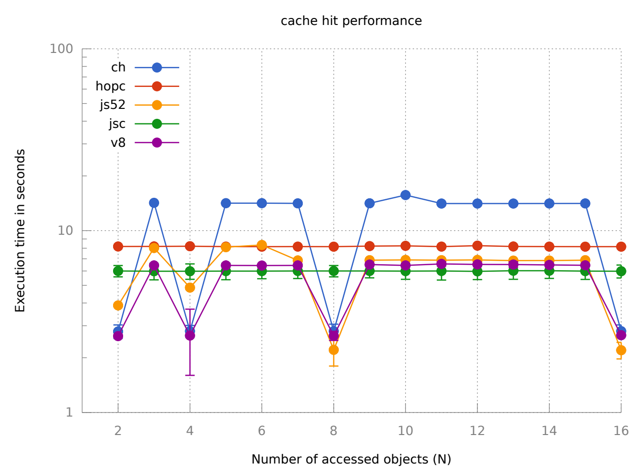

Figure 11: Cache hits maximal performance.

We started by measuring the optimal performance each system can deliver with 16 objects all sharing the same hidden class, that is, when `os` is defined as:

```
os = [{x: 21, y: 31}, {x: 12, y: 123}, ..., {x: 6, y: 1}]
```

With this setting, all accesses hit the cache and optimal performance is reached. These collected times will be reference times of each system with which we will compare the various proposed techniques when object definitions change. Results are reported in Figure 11. The horizontal axis is the value of the parameter `N`, which controls how many of the 16 objects are actually used. When `N` is 1, only the first of the `os` objects is accessed, when `N` is 2, the first and second objects are accessed, etc.

On this test, V8, Js52, and Ch, show surprising execution differences when `N` varies. As all reads hit the cache, flat horizontal plots would be expected for all systems. After narrowing down the issue, we eventually understood that this is due to the computation of the expression `i % N` that is used to rotate the objects in the read access (Figure 2, line 9). The execution of the remainder operation depends on the actual value of its divisor [Warren 2002, p. 266]. To eliminate this effect, we have modified the test to tabulate the modulo operations. This slows down all executions but also makes them steadier for all implementations but Js52. This is presented in Figure 12, top-left plots.

6.2 System Comparison

We have measured the performance of the caching techniques in all the situations described in this paper: monomorphic accesses, polymorphic/megamorphic accesses, getter accesses, and prototype accesses. All these are shown in Figure 12.

For the **monomorphic** tests, the 16 objects share the same hidden class and all accesses hit the cache. We see that all systems are in a range of 2s to 4s. Various reasons might explain the performance differences, for instance some systems might unroll the main loop while others, for instance Hopc, do not. These differences are unimportant for the rest of the evaluation as they are unrelated to the object accesses but these measures are useful to estimate the degradation of each system when execution contexts become more complex.

For the test **polymorphic/megamorphic** all the objects have a different hidden class. Then, when `N` is 2, two different objects are accessed, when `N` is 3, three different objects are accessed, etc. For the first two objects, Hopc uses the `imap` and `cmap` entries. It starts using vtables only when three different objects are used. When `N` increases, the frequency of `imap` and `cmap` hits decreases and vtables are used more and more frequently. As they require one additional read, the general performance decreases slowly. V8, Jsc, and Ch show no performance penalty for small `N` values (2 for V8 and 5 for Jsc and Ch). This could be explained by a loop unrolling that turns polymorphic accesses into monomorphic ones. Past that limit the slowdown of V8 and Ch is severe (about 13 times slower for V8 and 20 times for Ch). Js52 and Hopc delivers similar performance for this test but it must be noted that the slowdown Hopc imposes compared to monomorphic accesses is in the range 1.5 to 2, while for Js52 it is of more than 3. The technique Hopc uses to implement polymorphic/megamorphic accesses is then *relatively* more efficient than those of Js52.

For the test **accessor properties** the `os` objects belong to the same hidden class but the `x` properties are defined by 16 different accessors. Js52 and Jsc probably inline that accessor when only one object is used (`N=1`). We observe that Js52 and Ch pay a high price as soon as two objects are used. Jsc delivers stable performance up to `N` is 5. As already suspected, this could be explained by Js52 unrolling the loop 5 times, an optimization independent and potentially complementary to the technique used for inline caches. The other systems face a slowdown much more important than Hopc does as it outperforms all systems but Jsc when at least two objects are used. For V8, it should be noted that as for the polymorphic/megamorphic test, we observe a performance degradation when `N=5`, which might also confirm the hypothesis of a loop unrolling optimization.

For the **prototype accesses** the parameter `N` controls the number of accessed object *and* the length of the prototype chain used to obtain the property. When `N` is 1, the property is in the object, when it is 2, it is in its direct prototype, when it is 3, it is in the prototype of its prototype, etc. Only V8 and Hopc have performance independent of the prototype chain length. Jsc outperforms Hopc up to `N` equal 8. That is, Jsc pays no penalty for accessing properties in the prototype chain when the chain is small. Passed that threshold the performance degrades severely.

In conclusion, by comparing the optimal performance and the degraded performance of all systems we can establish that the techniques we propose enable Hopc to alleviate the impact of cache misses more effectively the all other systems. This is observed on the three situations considered in this study. In the next section we evaluate the practical impact of these optimizations on more realistic programs.

6.3 General Performance Evaluation

Selecting realistic JavaScript programs has long been identified as a difficult task [Ratanaworabhan et al. 2010] and is still today controversial. JavaScript is ubiquitous. It has been designed to run on the web browser but since some time it is also used extensively to program the server-side of web applications. Today, various operating systems use it as a scripting language (iOS, GnomeJS, ...). The current trend is to also use it for programming IoT applications. Designing a JavaScript benchmark suite representative of all these possible usages is difficult. In this study, we have collected a set of programs coming from the Octane, SunSpider, JetStream, Shoutout, Webtooling test suites from which we have filtered out those that are browser only and floating point intensive programs. Hopc does not optimize them yet and on these tests, execution times are dominated by garbage collection times, which makes them inappropriate to evaluate the impact of the inline caches described in this paper. To this list, we have added the adaptation of some programs implemented in other dynamic languages. These programs are generally middle-size programs composed of a single module, with the exceptions of `babylon`, which is 10.000 lines long, and `js-beautify`, which is a 22.000 line long multi-module program.

We have measured the acceleration obtained with enhanced property caches over plain property cache. That is, we have collected the execution times of plain property caches (as described in Section 2) and execution times of the enhanced caches presented in this paper. This is presented in Figure 13. First, we observe that for no benchmark the performance degrades when enhanced caches are used. Second, for all benchmarks but `tagcloud` we observe a significant speedup. The minor slowdown observed on `tagcloud` is not reproducible on other machines so we suspect it is due to a bad cache alignment on the particular computer and the particular OS version used for that experiment. Finally, we also observe highly different performance impacts. For instance, the benchmark `boyer` seems to only benefit from object extensions, while the `richards` benchmark mostly benefits from the vtable enhancement.

To better understand the reason for these different behaviors, we have collected numerical values about cache hits and cache misses. They are synthesized in Figure 14. The table reports for

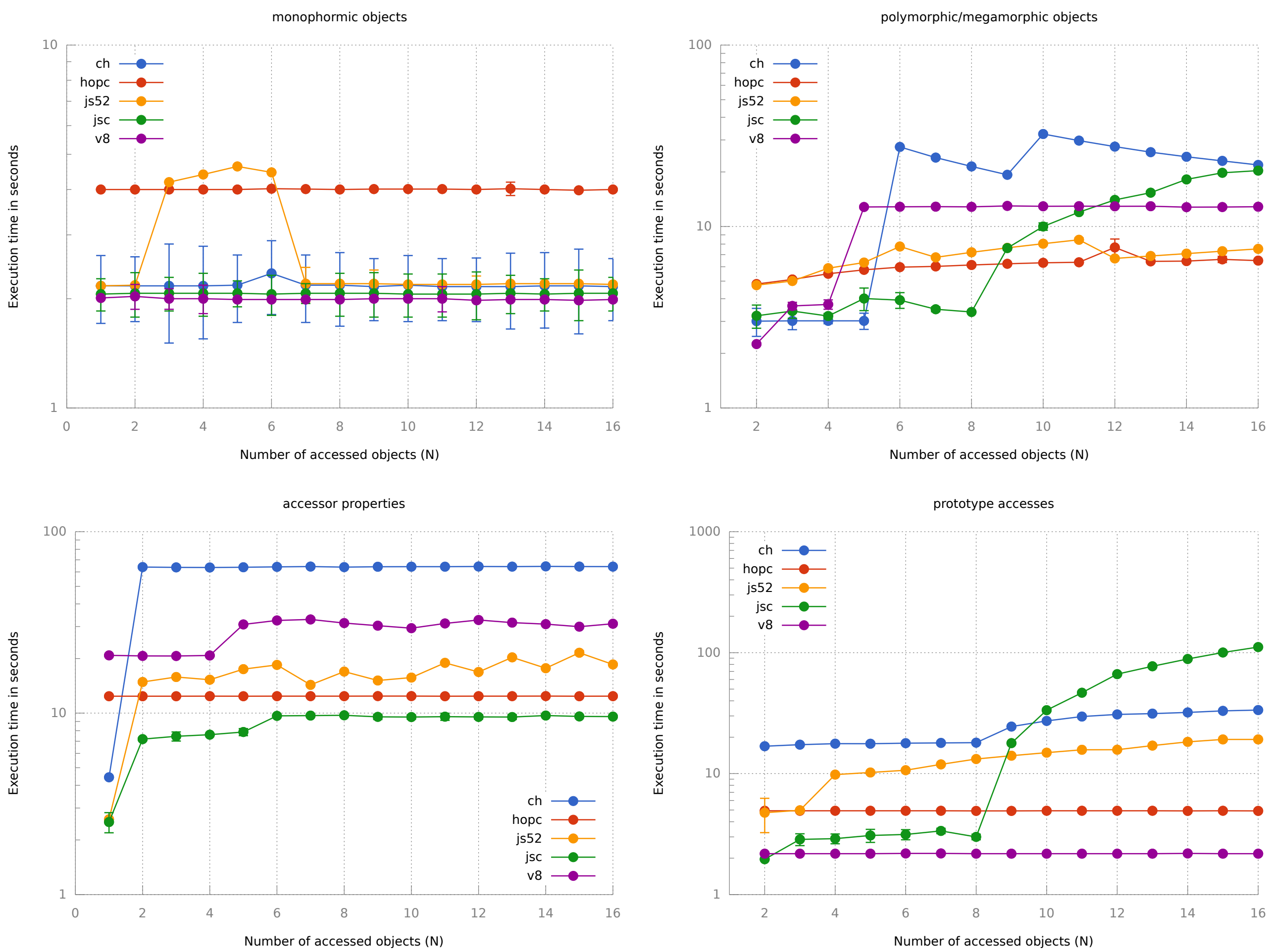

Figure 12: Cache hits performance. The vertical axis is the median of the wall clock execution times. Smaller is better.

each benchmark the number of cache hits and cache misses and the nature of the cache hits.

None of the benchmarks use property accessors. So, acache statistics are not included in Figure 14. This might either mean that optimizing accessors (Section 3.1) is useless or that the benchmarks do not cover sufficiently all JavaScript aspects. We opt for the second hypothesis as we are aware of real programs that use property accessors (for instance, the internal IO implementation of Node.js does use accessors). However, we have failed to find standard benchmarks that use them. As accessor properties impose only a minimal overhead (an extra test and a small code expansion) that JIT compilers and profile based compilation can eliminate, we think they are worth including in the arsenal of compiler optimizations.

The first observation that can be drawn from Figure 14 is that combining all the cache entries eliminate *all* cache misses for *all* accesses. The second observation is that the large majority of accesses need only one sort of cache entry. Only `deltablue-oo` uses many multiple cache entries per access sites. This means in the general case, the long cache probe sequence can be avoided. This is confirmed by the next experiment. We have measured the impact on the code sizes of caches. This is presented in Figure 15. We have measured the object file sizes produced by the compiler when no caches are used (the "no cache" column), when the complete cache sequence is used ("full cache"), and when the sequence is reduced using the profiling information a first execution has provided ("profile cache"). The smallest size is obviously observed when no caches at all are used. This is expected as the only code generated is the call to the cache miss routine. The figure shows the substantial gain the profile information enables. JIT compilers can expect the same benefit.

We have collected statistics about the virtual tables. Figure 16 shows the number of virtual tables created per benchmark, the memory space these tables occupy, and the maximal number of entries they contain. Benchmarks that use no virtual tables have been omitted from that table. It shows that very few accesses are megamorphic but when they are, virtual tables handle them efficiently. This experiment also shows that virtual tables globally use little memory.

benchmark	Hopc-plain	Hopc-proto	Hopc
acorn$^{\bullet}$	18.08 2.6%	17.87 1.9%	4.54 3.4%
babylon$^{\bullet}$	12.24 2.3%	12.17 1.6%	4.10 2.0%
bague*	0.50 3.1%	0.49 2.2%	0.50 4.2%
base64°	3.10 3.3%	3.11 3.4%	3.11 3.0%
binary-tree	59.87 3.7%	17.28 7.9%	16.86 2.5%
boyer†	12.19 3.4%	3.62 3.1%	3.58 0.9%
crypto-aes*	2.58 1.0%	2.58 2.6%	2.55 2.3%
crypto-md5*	0.60 2.9%	0.59 2.2%	0.59 3.4%
deltablue†	157.87 2.3%	57.01 2.6%	39.63 2.2%
deltablue-oo$^{\diamond}$	20.60 2.3%	8.72 3.0%	6.19 4.4%
earley†	2.75 2.0%	2.39 2.0%	2.36 1.9%
fannkuch$^{\diamond}$	3.69 0.6%	3.66 0.9%	3.66 0.5%
js-beautify$^{\diamond}$	4.50 2.4%	4.09 0.9%	3.81 1.2%
maze*	1.79 2.4%	1.20 2.9%	1.20 3.1%
puzzle*	2.68 0.8%	2.72 0.5%	2.70 0.4%
qsort*	0.83 2.0%	0.84 2.6%	0.83 2.0%
richards†	59.23 1.7%	54.19 3.0%	16.01 1.1%
sieve*	5.76 1.6%	2.50 2.2%	2.49 2.5%
splay†	9.88 4.6%	9.38 5.2%	9.21 5.4%
tagcloud*	7.39 10.0%	7.73 4.3%	8.04 5.4%

Figure 13: Results of 30 runs collected on an Intel Xeon E5-1650 running Linux 4.17/Debian. Median of wall clock times and relative standard deviation are shown. Smaller time is better. Hopc-plain shows execution times with traditional caches. Hopc-proto shows execution times when all (Section 3) but vtables are activated. Hopc shows the execution times when all property caches enhancements are enabled. Benchmark sources: † Octane, * Jetstream, $^{\circ}$ Sunspider, * Bglstone, $^{\bullet}$ Webtooling, $^{\diamond}$ other.

7 RELATED WORK

JavaScript performance has dramatically improved over the years. The fastest implementations are due to major industrial actors, namely Google, Mozilla, Apple, and Microsoft. Some parts of these implementations are described in more or less formal blogs [Apple 2018; Google 2018; Microsoft 2018; Schneider 2012; Wingo 2013]. Some academic publications also document these systems [Gal and *et al.* 2009; Hass and *et al.* 2017].

Ahn *et al.* studied the impact of V8 object representation in the context of web client-side programs [Ahn et al. 2014]. First they measured the impact of inline caches on various benchmarks. They show how important this technique is and they also show that the impact of property caches is highly dependent on the nature of the programs themselves. Server-side programs appear to be much more beneficial than client-side programs (in their study, server-side programs are represented by the classical JavaScript benchmark suites Octane, Sunspider, and Kraken, and client-side programs are represented by jsmeter [Ratanaworabhan et al. 2010]). This is because client-side programs break more frequently the assumptions that prototypes and method bindings almost never change during executions. Then, the authors of this study suggest to extract the prototype links and the method bindings from the hidden classes, which would break the fast prototype chain access presented Section 3. They present an experimental report that shows client-side improvements but it also shows server-side slowdowns. This does not contradict our own results.

The blog article [Bevenius 2018] describes the polymorphic inline caching optimization recently added to V8. This optimization is similar to the one of the SELF system [Hölzle et al. 1991]. The blog [Bruni 2017] describes the latest V8 property accesses implementation. It presents a mostly standard property cache implementation without any details about prototypes, polymorphic accesses, or megamorphic accesses. It shows the object memory models that support *inline* and *external* properties. Contrary to Hopc, a V8 object might simultaneously contain inline and external properties. Determining the impact of the two strategies is beyond the scope of the present study that focuses on cache implementation only.

Clifford and his colleagues have proposed a dynamic setting to keep track more accurately of object sizes [Clifford 2015]. This enables more efficient allocations and increases the number of properties that are stored inline. Hopc uses a similar technique. Although strongly connected, the techniques that focus on the object memory layouts are not strictly related to the property cache management studied in this paper.

The virtual tables presented in Section 4 follows a long tradition of work on fast property accesses and fast type checks for object-oriented languages. The loose structuring of prototype-based object orientation makes most studies unsuitable. Techniques developed for single inheritance testing [Cohen 1992] are inapplicable because the prototype chaining enables complex inheritance hierarchies. Techniques developed for multiple inheritance [Alpern et al. 2001; Ducournau and Morandat 2011] hardly apply because of the dynamic context in which JavaScript programs execute that is liable to create new classes at any time. The originality of Hopc for dealing with polymorphism consists in attaching the virtual tables to the access point locations instead of attaching them to the object classes.

The storage strategies developed for optimizing the representations of homogeneously typed collections [Bolz et al. 2013] complements the following studies as it focuses on the memory layout and memory organization of objects.

The paper [Gong et al. 2014] presents a profiler for JavaScript that after a source-to-source transformation pinpoints *JIT-unfriendly* code. The profiler is used to track property cache misses but it only detects inconsistent object layouts, that is objects owning the same properties but belonging to different hidden classes. Inconsistent layouts are treated naturally by the vtables mechanism but as their analysis shows many tests suffer from this problem, it might suggest that the compiler should be provided with special optimizations for removing them. Constructors could be easily optimized using a static analysis similar to the one used to estimate object sizes. It could sort the properties and use that sorting for creating consistent object memory layouts.

8 CONCLUSION

We have presented several techniques that complement and enhance property caches used for accessing object properties of JavaScript like languages. They take over classical caches when the searched property is either stored in an object of the prototype chain or defined using accessors. They also support efficiently polymorphic and megamorphic property accesses. Finally, they also support efficient object extensions. These techniques do not apply as frequently as simple property caches that cover a vast majority of accesses. However, since they impose no overhead when not used,

	accesses		*multi*	*inv*	*cached proto accesses*		*vtable accesses*		
benchmark	get+put+call	cache misses	all	all	get	put	get	put	call
acorn	36×10^6	849×10^3 (3%)	58%	10×10^3	4×10^6 (15%)	3×10^6 (36%)	414×10^3 (2%)	217×10^3 (4%)	0 (0%)
babylon	26×10^6	2×10^6 (5%)	77%	11×10^3	3×10^6 (12%)	2×10^6 (31%)	641×10^3 (4%)	134×10^3 (4%)	0 (0%)
bague	~ 0	0 (0%)	0%	125	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
base64	17×10^6	47 (0%)	0%	129	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
binary-tree	2×10^9	56 (0%)	0%	126	303×10^6 (34%)	606×10^6 (100%)	0 (0%)	0 (0%)	0 (0%)
boyer	463×10^6	175 (0%)	0%	131	0 (0%)	103×10^6 (100%)	0 (0%)	0 (0%)	0 (0%)
crypto-aes	9×10^6	8×10^3 (0%)	0%	129	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
crypto-md5	~ 0	0 (0%)	0%	129	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
deltablue-oo	2×10^9	26×10^6 (2%)	26%	178	5×10^6 (0%)	17×10^6 (16%)	0 (0%)	3×10^6 (3%)	0 (0%)
deltablue	17×10^9	233×10^6 (2%)	26%	178	5×10^9 (32%)	148×10^6 (16%)	0 (0%)	25×10^6 (3%)	0 (0%)
earley	10×10^6	3×10^3 (0%)	0%	131	0 (0%)	5×10^6 (89%)	0 (0%)	0 (0%)	0 (0%)
fannkuch	~ 0	0 (0%)	0%	131	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
js-beautify	47×10^6	17×10^3 (0%)	9%	297	7×10^6 (16%)	2×10^6 (43%)	0 (0%)	0 (0%)	0 (0%)
maze	72×10^6	2×10^6 (2%)	0%	141	7×10^6 (10%)	7×10^6 (77%)	0 (0%)	0 (0%)	0 (0%)
puzzle	~ 0	0 (0%)	0%	125	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
qsort	~ 0	0 (0%)	0%	129	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
richards	11×10^9	146×10^6 (2%)	0%	151	2×10^9 (19%)	3×10^6 (0%)	0 (0%)	0 (0%)	0 (0%)
sieve	102×10^6	19 (0%)	0%	132	0 (0%)	52×10^6 (100%)	0 (0%)	0 (0%)	0 (0%)
splay	265×10^6	2×10^3 (0%)	7%	135	10×10^6 (5%)	14×10^6 (19%)	2×10^6 (0%)	0 (0%)	0 (0%)
tagcloud	56×10^6	2×10^3 (0%)	0%	129	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)

Figure 14: Cache statistics. The two first columns report the global number of hits and misses. The direct cache hits (cmap hits) are not reported in this table because they correspond to standard JavaScript implementations. These numbers can still be deduced by subtracting the number of misses from the number of hits. The *multi* column shows the number of caches that use more than one single map. The *inv* column shows the number of `pclass` invalidations.

benchmark	no cache	full cache	profile cache
acorn	1645kb	3068kb	2900kb
babylon	3032kb	5686kb	5438kb
bague	27kb	29kb	29kb
base64	37kb	38kb	38kb
binary-tree	38kb	67kb	57kb
boyer	772kb	1139kb	922kb
crypto-aes	103kb	107kb	107kb
crypto-md5	111kb	111kb	111kb
deltablue-oo	212kb	409kb	394kb
deltablue	417kb	1056kb	937kb
earley	713kb	1020kb	834kb
fannkuch	29kb	29kb	29kb
js-beautify	2551kb	4095kb	3916kb
maze	192kb	404kb	376kb
puzzle	35kb	38kb	38kb
qsort	28kb	30kb	30kb
richards	234kb	555kb	505kb
sieve	33kb	40kb	35kb
splay	99kb	195kb	143kb
tagcloud	293kb	350kb	350kb

Figure 15: Object file sizes depending on the compilation mode. *No cache* uses no inline cache at all. *Full cache* corresponds to object sizes with the full cache check sequence generated. *Profile cache* corresponds to object file sizes after profile-guided optimization.

they can be integrated in any existing system at no run time cost. We have validated the approach with an experimental report based on Hopc, an AOT JavaScript compiler. It shows that the presented techniques improve performance in situations where simple cache miss.

benchmark	# vtables	memory	# locations	poly
deltablue	23	1kb	6	7
deltablue-oo	22	1kb	6	7
splay	1	0kb	1	1

Figure 16: Vtable statistics. The column "# vtables" reports the total number of vtables use by a benchmark. The column "memory" gives information about the overall vtable memory footprint. The column "# locations" reports the number of access points that use vtables, and "poly" reports the maximum number of entries contained by vtables.

REFERENCES

W. Ahn, J. Choi, T. Shull, M. Garzarán, and J. Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14), Edinburgh, Ireland.*

B. Alpern, A. Cocchi, S. Fink, and D. Grove. 2001. Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*. USA, 108–124.

Apple. 2018. WebKit. https://webkit.org/blog/. (2018).

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

D. Bevenius. 2018. Learning Google V8. https://github.com/danbev/learning-v8. (2018).

C. F. Bolz, L. Diekmann, and L. Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. 167–182.

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

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.

D. *et al.* Clifford. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. USA, 105–117.

N. Cohen. 1992. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 4 (1992), 626–629.

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.

R. Ducournau and F. Morandat. 2011. Perfect class hashing and numbering for object-oriented implementation. *Software: Practice and Experience* 41, 6 (2011), 661–694.

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. Gal and *et al.* 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. 465–478.

L. Gong, M. Pradel, and K. Sen. 2014. *JITPROF: Pinpointing JIT-unfriendly JavaScript Code*. Technical Report UCB/EECS-2014-144.

Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 357–368. DOI: http://dx.doi.org/10.1145/2786805.2786831

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

A. Hass and *et al.* 2017. Bringing the Web Up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, USA, 185–200.

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)*. UK, 21–38.

Microsoft. 2018. Chakra. https://github.com/Microsoft/ChakraCore/wiki/Resources. (2018).

P. Ratanaworabhan, B. Livshits, and B. G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps'10)*. USA, 12.

F. Schneider. 2012. High JavaSscript performance with V8. http://cs.au.dk/~jmi/VM/IC-V8.pdf. (March 2012).

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

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

Henry S. Warren. 2002. *Hacker's Delight*. Addison-Wesley, USA.

A. Wingo. 2013. Optimizing Let In SpiderMonkey. https://wingolog.org/archives/2013/12/18/optimizing-let-in-spidermonkey. (Dec. 2013). https://wingolog.org/archives/2013/12/18/optimizing-let-in-spidermonkey

A. Wöß and *et al.* 2014. An Object Storage Model for the Truffle Language Implementation Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, USA, 133–144.