# Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems

Tomoharu Ugawa
tugawa@acm.org
University of Tokyo
Japan

Stefan Marr
s.marr@kent.ac.uk
University of Kent
United Kingdom

Richard Jones
R.E.Jones@kent.ac.uk
University of Kent
United Kingdom

## Abstract

JavaScript is increasingly used for the Internet of Things (IoT) on embedded systems. However, JavaScript's memory footprint is a challenge, because normal JavaScript virtual machines (VMs) do not fit into the small memory of IoT devices. In part this is because a significant amount of memory is used by hidden classes, which are used to represent JavaScript's dynamic objects efficiently.

In this research, we optimize the hidden class graph to minimize their memory use. Our solution collects the hidden class graph and related information for an application in a profiling run, and optimizes the graph offline. We reduce the number of hidden classes by avoiding introducing intermediate ones, for instance when properties are added one after another. Our optimizations allow the VM to assign the most likely final hidden class to an object at its creation. They also minimize re-allocation of storage for property values, and reduce the polymorphism of inline caches.

We implemented these optimizations in a JavaScript VM, eJSVM, and found that offline optimization can eliminate 61.9% of the hidden classes on average. It also improves execution speed by minimizing the number of hidden class transitions for an object and reducing inline cache misses.

***CCS Concepts:*** • **Software and its engineering → Virtual machines**.

***Keywords:*** JavaScript, virtual machine, embedded systems, hidden class, inline caching, profiling, offline optimization, IoT

**ACM Reference Format:**
Tomoharu Ugawa, Stefan Marr, and Richard Jones. 2022. Profile Guided Offline Optimization of Hidden Class Graphs for JavaScript VMs in Embedded Systems. In *Proceedings of the 14th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '22), December 05, 2022, Auckland, New Zealand*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3563838.3567678

*VMIL '22, December 05, 2022, Auckland, New Zealand*


## 1 Introduction

Traditionally, JavaScript has been used for web applications. However, its event-driven programming style and support for rapid program development has led to interest in deploying JavaScript for Internet of Things (IoT) applications on embedded systems. Notable examples include Espruino,[1] IoT.js,[2] and Moddable[3] [8]. Unfortunately the huge memory footprint of modern JavaScript virtual machines (VMs) precludes their use in resource constrained embedded systems. The Raspberry Pi Pico micro controller, a typical micro controller for IoT, has only 256KB SRAM [4]. Thus, embedded systems require a specialized JavaScript VM that represents objects and their associated data structures in a space efficient form.

Fortunately, such JavaScript VMs specialized for a predetermined program are practical for embedded systems. In contrast, JavaScript VMs for browsers must be able to execute any JavaScript program found in the web pages users access. Some JavaScript VMs already do this specialization in embedded systems [8, 17] and even Java server applications have started to adopt support for such closed-world approaches with GraalVM's native-image compilation [18]. The closed-world assumption brings the opportunity to apply offline profile-guided optimization (PGO) to the VM. We can execute the program with a profiling VM to collect information about its behaviour and then optimize the VM for this particular program.

Modern JavaScript VMs usually use an optimization referred to as hidden classes or maps [3], which represent JavaScript's dynamic objects efficiently. The hidden classes are meta-objects that represent objects' layout information, i.e., property names and offsets to their values. Hidden classes are created on-demand and cached, forming a state transition graph called the *hidden class graph*, whose nodes are hidden classes representing a state of the object. Whenever a property is added to an object, a new hidden class is allocated

[1] https://espruino.com
[2] https://iotjs.net
[3] https://www.moddable.com/
[4] https://www.raspberrypi.com/products/raspberry-pi-pico/

(unless one already exists) and added to the graph, and the state of the object *transitioned* to use this hidden class.

Although hidden classes represent objects efficiently, they come at some cost. First, they are cached and occupy memory, enlarging an application's memory footprint. Second, hidden class transitions may need to allocate new hidden classes and to re-allocate the storage for property values.

In this research, we are the first to apply offline PGO to hidden classes. More specifically, we collect the hidden class graph and related information for each allocation site in a profiling run. Based on the profiling results, we construct an optimized hidden class tree *offline*, with the following novel optimizations:

- We compute the hidden class that most objects allocated at a site will have.
- We eliminate *intermediate* hidden classes and transitions, which are common when properties are added one after another.
- We bias the chains of hidden class transitions to favor the most frequently used case.

The optimized hidden class graphs are loaded in the actual run, and installed at allocation sites. Thanks to the optimized graph, objects can be created *pretransitioned* to the first hidden class in the optimized graph and with sufficient storage for property values, avoiding the need for re-allocation. Furthermore, eliminating intermediate hidden classes reduces the variation of hidden classes, thus helping to reduce the polymorphism of the inline caches.

We implemented these offline optimizations of the hidden class graph in eJSVM [17], a JavaScript VM for embedded systems. These optimizations improved the memory footprint of all our benchmark programs, reducing the volume of hidden classes by 61.9% on average and overall footprint by 3–30%. Execution speed improved for all benchmarks, in the best cases by 11.6–17.6%. This improvement came from not only avoiding re-allocations but also improvements in the inline cache hit ratio.

The contributions of this work are as follows:

- We demonstrate that the hidden class graph is reusable across program runs.
- We propose offline optimizations for hidden class graphs.
- We implement and evaluate several offline optimizations in a JavaScript VM for embedded systems and demonstrate that the memory footprint is reduced while execution speed is improved.

## 2 Background

***JavaScript.*** JavaScript is a dynamic programming language. Non-primitive data in JavaScript are called objects, which correspond to mutable mappings from property names to their values. Thus, a JavaScript program can add arbitrary properties to objects after they are created. For example, the program in Figure 1 creates an object without properties. Then, it adds properties `x` and `y` one after another.

In addition to properties, JavaScript's objects have *prototype objects*. When a program tries to read from a property that the object does not have, it looks up the prototype object for the property value. Prototype objects have in turn their own prototype objects. A read request for a property that the prototype object does not have is forwarded to its prototype object. In this way, prototype objects form a chain, called the *prototype chain*.

For example, the program in Figure 2 creates an object `a` with the prototype `{x:0, y:0}`, which is set to the `prototype` property of the constructor function `Point`. Then, the program adds a property `x` to the object. When reading from the property `x` subsequently, the value is obtained from the `a`'s own property. In contrast, reading from the property `y` yields the value of the prototype object's property, because the object `a` does not have its own property `y`. Neither `a` nor its prototype have the property `z`, and there is no further prototype object above `a`'s direct prototype, so `z` is `undefined`.

***Hidden Classes.*** In a straightforward implementation, JavaScript's objects could be represented by association lists or hash tables. Such an implementation would allow each object to have a different set of properties at the same cost as when all objects have the same set of properties. However, realistic JavaScript programs tend to give the same properties to multiple objects. For example, objects initialized with the same constructor function are likely to have the same properties. This brings an opportunity for objects to share layout information.

The hidden classes are meta-objects that represent layout information of objects. With hidden classes, an object has a pointer to its hidden class and an array of property values called the *property array*. A hidden class has a mapping from property names to their indexes in the array. Figure 3 shows the object `a` in Figure 1 represented using a hidden class.

```
a = {};
a.x = 10;
a.y = 20;
```

**Figure 1.** A JavaScript object `a` is created without properties; `x` and `y` are then added afterwards.

```
Point.prototype = {x: 0, y: 0};
a = new Point();
a.x = 10;
// a.x = 10, a.y = 0, a.z = undefined
```

**Figure 2.** Property accesses may be delegated to a prototype. Here `x` and `y` are defined in the prototype object of `a`. Then, a new property `x` is defined directly on `a`.

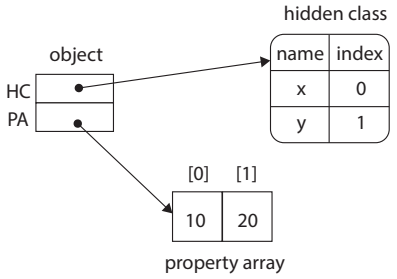


**Figure 3.** Object a with its hidden class and the property array for its property values.

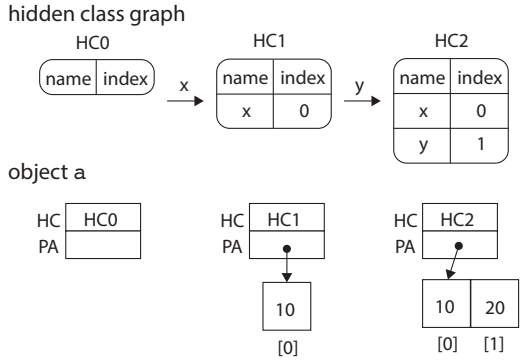


**Figure 4.** Hidden class graph for object a, with the hidden class first empty, and then with property x, and finally with the property y added.

Hidden classes are shared by objects with the same layout. Thus, hidden classes are immutable. Whenever a new property is added to an object, the runtime must find the next hidden class that has the added property. If such a hidden class has not been created, the runtime creates and caches it.

To find next hidden class quickly, the hidden classes are organized as a state transition graph. The directed edges of the hidden class graph are labelled with property names, each of which leads to the next hidden class when the property with that name is added.

Figure 4 shows the hidden class graph for the program in Figure 1 along with the evolution of the form of object a. Because object a is created with no properties, initially it has an empty hidden class HC0 and no property array. When property x is added, the object is given a new hidden class HC1, or *transitions* to HC1. At the same time, a property array with one slot is allocated to store the value for x. Then, when property y is added, it further transitions to HC2, and expands the property array to store the value for y. In the rest of this paper, we denote a hidden class like an object with a mapping. For example, we denote HC2 as `{x:0, y:1}`.

Hidden class graphs for larger programs usually have branches. Figure 6 shows the hidden class graph created after executing the function `figure` in Figure 5 multiple

```
function figure(is_circle) {
  s = {}       // s.HC: {}
  s.x = 10;    // s.HC: {x: 0}
  s.y = 20;    // s.HC: {x: 0, y: 1}
  if (is_circle) {
    s.r = 5;   // s.HC: {x: 0, y: 1, r: 2}
  } else {
    s.w = 20;  // s.HC: {x: 0, y: 1, w: 2}
    s.h = 15;  // s.HC: {x: 0, y: 1: w: 2, h: 3}
  }
  s.c = "red"; // s.HC: {x: 0, y: 1, r: 2, c: 3} or
               //       {x: 0, y: 1: w: 2, h: 3, c: 4}
  return s;
}
```

**Figure 5.** A program yielding a branching hidden class graph as different properties are added as `figure` is invoked with different values of `is_circle`.

times with different `is_circle` flags. In this figure, `s.HC:` in comments indicates the mapping of the hidden class of `s` on this line. Suppose that `figure` is called with `true` in the first invocation. In this invocation, an empty object `s` is created, and properties `x`, `y`, `r`, and `c` are added one by one. As properties are added, the hidden classes HC0, HC1, HC2, HC3, HC4 are created and the object transitions along the path in the hidden class graph. Also suppose that `is_circle` is `false` in the second invocation. This time, properties `w` and `h` are added instead of `r`. The object transitions from HC0 to HC1 and HC2 using the cached hidden classes. When property `w` is added, the runtime fails to find the next hidden class. Thus, HC5 is created and attached to HC2 in the hidden class graph, creating a branch.

In this example, property `c` is added as the last property regardless of `is_circle`. However, HC4 and HC7 are different because properties in the middle are different. Even if two hidden classes had the same set of properties names, their offsets might differ. Thus, they cannot generally be combined into a single hidden class.

***Problem with Hidden Classes.*** The first problem is that the hidden classes may occupy a non-negligible fraction of the heap. Furthermore, hidden classes are often kept cached for future use. For example, we observed 21–49KB of hidden-class-related data in eJSVM for the Are We Fast Yet benchmarks [13] (see Figure 13). Compared to the 256KB of SRAM in embedded systems, these numbers are not negligible.

To mitigate this problem, the V8 engine implements hidden classes with association lists. This allows hidden classes to share common mappings with their predecessors. For example, the mapping `x:0` can be shared among HC1 to HC7 in Figure 6. Truffle creates the mapping part of hidden classes lazily; it creates a hidden class without mapping, and only constructs the mapping, by collecting mapping information

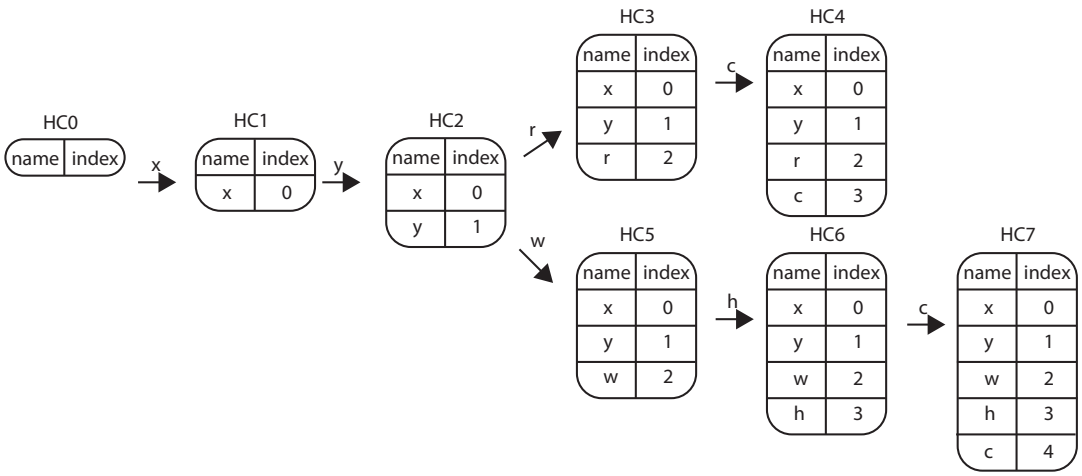


**Figure 6.** The hidden class graph for the code in Figure 5. The branch in the graph comes from different properties being added depending on the value of the `is_circle` flag.

from the edge labels on the path to the hidden class, when the mapping is used for the first time [20].

The second problem is that the property arrays are re-allocated whenever a new property is added to an object, as shown in Figure 4. When an object is created, we do not know how many properties are eventually added to the object. Thus, we have to expand the storage of property values on demand. This requires the storage of property values to be separated from the object in a property array so that we can re-allocate it. Frequent re-allocation degrades performance because it requires copying memory and increases garbage collection (GC) frequency. We could speculatively allocate space for future properties, but allocating too much space enlarges the memory footprint. Truffle reserves space for a fixed number of properties when an object is created.

V8 mitigates this problem by a technique called slack tracking,[5] which guesses the final size of the object being allocated from past behaviour of the program. V8 first allocates a large amount of extra space for an object. Objects are assumed to reach their stable state after executing for a while. Then, it memorizes the number of properties of stable objects so that objects created by the same constructor can be created with storage to contain exactly the same number of properties as objects in their stable state. If stable state objects created by the same constructor have different numbers of properties, the larger number is used, trading space for speed.

Clifford et al. also mentioned *pretransitioning* for V8 [6]. They collect stable states of objects and relate them to their allocation sites. To do so, they attach mementos to objects. Mementos are short-lived objects that keep track of allocation sites until the attached objects are promoted to the old space by the GC.

[5] https://v8.dev/blog/slack-tracking

The property storage can be allocated *in* the object if its size is known when the object is created. Properties placed in the objects are called *in-object properties*. If more properties than the storage for in-object properties are added, overflowing properties are placed in the property array.

***Inline Caching.*** When a program accesses a property of an object, the runtime first resolves the property name to obtain the index corresponding to that name from the hidden class. Then, it accesses the slot in the property array. Accessing the property array by an index is efficient. However, resolving the name is not; typically it involves a hash table or an association list look up.

Inline caching [7] is a common optimization that accelerates property accesses. An inline cache stores the resolved index at the program location where the property is accessed, i.e., the *access site*. When the runtime resolves a name for the first time at an access site, it caches the hidden class and the index. For later accesses at the same access site, the runtime then merely checks that the same name and hidden class are used and then uses the cached index.

In JavaScript, access sites tend to be polymorphic; objects reaching the same access site tend to have various hidden classes. This is because, first, hidden classes change as properties are added. Second, the hidden class graph may branch as shown in Figure 6; objects with hidden class HC4 and HC7 may come to the same access site accessing `x`, for example. Thus, performance-oriented VMs employ *polymorphic inline caching* [10], where each access site has multiple cache entries. An access site has a mapping from hidden classes to indexes, and the runtime looks up the index by the hidden class of the incoming object. Polymorphic inline caching specialized to class base inheritance of TypeScript is also proposed [19]. eJSVM does not currently support polymorphic inline caching.

```
p1 = {y: 3};         // p1.HC: {y: 0}
p2 = {x: 4, y: 7};   // p2.HC: {x: 0, y: 1}
C1.prototype = p1;
C2.prototype = p2;
a = new C1();        // a.HC: {}
a.u = 10; a.v = 20; // a.HC: {u: 0, v: 1}
b = new C2();        // b.HC: {}
b.u = 5;  b.v = 10; // b.HC: {u: 0, v: 1}
f(a)   // f accesses a.y
f(b)   // f accesses b.y
```

**Figure 7.** Objects with different prototype chains may have the same hidden class, as long as the prototype is not considered part of the hidden class. Thus, here a and b have the same hidden class, although a's prototype is p1 and b's is p2.

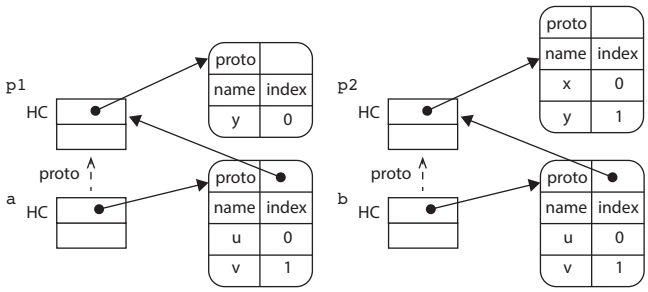


**Figure 8.** Final states of objects in Figure 7 in an implementation that puts prototype pointers in hidden classes. Dotted lines represent prototype relationships while solid lines represent actual pointers.

***Inline Caching for Prototype Properties.*** Inline caching for properties on the prototype chain requires caching the prototype object owning the property as well as the index. In addition, it requires a different guard. An inline cache for a prototype property is valid only if the prototype chain between the incoming object and the owner object of the property is the same. For example, a and b in Figure 7 obtain properties u and v in the same order. Thus, they have the same layout. Suppose that function f has a site accessing property y of the f's argument. In the invocation with a, y is found in its prototype p1, and its index is 0. Thus, the pair of owner and index, (p1, 0), is cached. However, in the invocation with b, y should be found in p2, and its index should be 1. Thus, the cache is invalid even if a and b have the same hidden class.

To quickly confirm that the prototype chain is the required one, we can put the pointer to the prototype object in the hidden class rather than in the object itself. Figure 8 shows objects a and b with their hidden classes having prototypes. As shown in this figure, objects with different prototypes have different hidden classes. Thus, we can guarantee the validity of the cache by testing the hidden class, as long as prototype objects on the chain do not change, i.e., no properties are added or removed and the prototype relation is not altered.

When a prototype object changes, the related inline caches need to be invalidated. V8 maintains the prototype relationship by organizing a tree of hidden classes to minimize the number of caches to be invalidated[6]. eJSVM invalidates all inline caches whenever a prototype object changes.

## 3 Optimizing the Hidden Class Graph

As previously outlined, when targeting an embedded system we can make strong assumptions about applications, for instance assuming a closed world. For these types of system, we can optimize the hidden class graph offline based on the object behaviour observed in a profiling run.

***Pretransitioning.*** In practice, object shapes tend to be predictable for specific allocation sites and allocation sites in turn are often *layout-monomorphic*, i.e., all objects allocated at a particular site eventually have the same layout. The allocation site in Figure 1 is an example.

By knowing the final layout for a layout-monomorphic allocation site, we can allocate a sufficiently large in-object property area and pretransition the newly created object to the hidden class representing its final layout. We can also eliminate the intermediate steps in the hidden class graph, keeping only the final hidden class in the graph for a specific allocation site. For example, the allocation site in Figure 1 only needs HC2 in Figure 4, optimizing out HC0 and HC1.

***Elimination of Intermediate Hidden Classes.*** Even for *layout-polymorphic* allocation sites, although objects allocated there eventually have different hidden classes, objects often share a common path in the hidden class graph.

For example, objects created in Figure 5 will eventually be of either hidden class HC4 or HC7. The chains of transitions leading to these final hidden classes share the same prefix, HC0→HC1→HC2 (see Figure 6). Furthermore, once the next property is added to an object in HC2, we can determine the final hidden class for the object. For example, if property r is added, the object will eventually transition to HC4.

With an optimized hidden class graph, we pretransition newly created objects HC2. Then, when either property r or w is added, we transition the object to HC4 or HC7, skipping the *intermediate* hidden classes HC3, or HC5 and HC6.

In general, for layout-polymorphic allocation sites we can construct the optimized hidden class graph by removing intermediate steps, except for branching nodes. This optimization reduces the number of transitions, and hence the number of re-allocations of property arrays.

[6] https://mathiasbynens.be/notes/prototypes

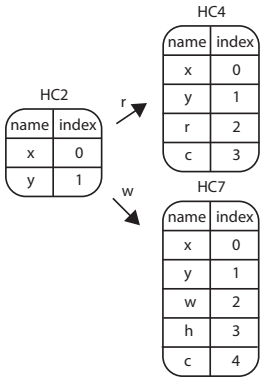


**Figure 9.** Optimized version of the hidden class graph of Figure 6.

In summary, these optimizations eliminate *intermediate* hidden classes (defined as those that have exactly one outgoing edge). After eliminating intermediate hidden classes, the hidden class graph in Figure 6 becomes the one shown in Figure 9.

***Preserving Hot Hidden Classes.*** Eliminating all intermediate nodes may yield suboptimal memory footprint in the following two cases. First, a large number of objects created at an allocation site may die before all properties are added. In this case, aggressive pretransitioning results in increased memory use, since memory is allocated for properties that will not be used.

Second, aggressive pretransitioning may result in memory being allocated too eagerly, possibly interfering with memory-intensive tasks. For example, the 10,000 objects created in Figure 10 eventually have the properties `x`, `y`, `u`, and `v`. However, initially only `x` and `y` are added and then `memory_intensive_task` is called. When creating these objects pretransitioned to their final hidden class `{x:0, y:1, u:2, v:3}`, memory is allocated for properties `u` and `v`, which are unused at this point, unduly inflating memory use, potentially increasing the maximum footprint and GC pressure.

For both cases, pretransitioning can increase memory use significantly depending on the number of objects and usage characteristics of slots, reducing the available memory for other tasks.

To address this issue, we identify *hot* hidden classes based on usage counts (see Section 4.1) and preserve them regardless of whether they are an intermediate hidden class or not. For the program in Figure 10, the hidden class `{x:0, y:1}` has a single transition, but is hot. Thus, we should preserve it to avoid wasting memory at the time when the peak memory usage is reached.

***Moving Branches.*** Programs may have a cold hidden class that is very similarity to a hot one. Here, it may be beneficial to merge them by moving branches in the hidden class graph to linearize the path to a hot hidden class.

```
ary = []
for (i = 0; i < 10000; i++) {
  a = {x: ..., y: ... };
  ary.push(a);
}
memory_intensive_task(); // peak usage
for (i = 0; i < 10000; i++) {
  ary[i].u = ...;
  ary[i].v = ...;
}
```

**Figure 10.** Peak of memory usage is marked in `memory_consuming_task`. Objects in `ary` are yet to have properties `u` and `v` at the peak. If their areas are allocated speculatively, the peak rises.

For the program in Figure 5, suppose that `is_circle` is very likely to be `true`. In this case, we speculatively create objects with hidden class HC4 rather than HC2. This is done by moving the branch with property `w` from HC2 to HC4, as shown in Figure 11. Then, we can avoid all transitions for the objects that eventually have HC4. The drawback is that we must give the storage for property `r` to those objects that have `w` and `h` rather than `r`, yielding a hidden class HC7′, different to HC7. Because we cannot guarantee that `r` and `w` are exclusively used in the general case, we cannot assign the storage for property `r` to property `w`. However, assuming that almost all objects have `r`, this optimization is practical. Note that because HC4 has property `c`, it is not added on the transition from HC4 to HC7′.

In general, we can optimize the hidden class graph to favor hot transition chains, which most objects take. More specifically, we can push cold branches towards the final hidden class of the hot transition chain, thus straightening the trunk of the chain. This branch moving involves reordering of properties of the hidden classes that appear after the moved branches; the transition on the moved branch and any further transitions must add properties that have not been added in the hot chain.

## 4 Implementation

Our hidden class graph optimizations are part of the offline optimizer of eJSVM. The optimization process consists of two steps. First, a *profiling VM* executes the target program to collect the hidden class graph and related information. Second, the *hidden class graph optimizer* uses this profile to construct an optimized hidden class graph and a list of allocation sites and their entry points in the graph, i.e., the hidden classes to be used to create objects. This optimized hidden class graph can then be loaded by the VM together with the target program for a more memory efficient execution.

***Allocation Sites.*** Since eJSVM is a bytecode interpreter, a JavaScript program is first compiled into bytecodes before

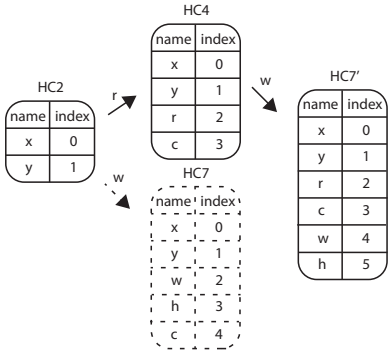


**Figure 11.** The optimized hidden class of Figure 9 after moving branches.

execution starts. Allocation sites are the `new` bytecodes that create empty objects. They correspond to JavaScript's `new` keyword and the use of object literals. Object literals are compiled into bytecode sequences including `new` and the operations to add the initial properties one by one.

Although some builtin functions allocate objects (e.g., `Object.create` and `Array.concat`), we do not currently consider them as allocation sites. Builtin functions may allocate multiple objects, each of which requires a hidden classes graph. Because allocation sites are related to bytecodes, the call bytecode is considered as the site allocating all objects in a builtin function. However, our current implementation can only associate at most one hidden class graph with each allocation site. Thus, we do not apply our optimization to builtin functions. These functions will create hidden classes dynamically as they would without our optimization.

### 4.1 Profiling

The profiling VM executes the target program and creates the hidden class graph dynamically, with a hidden class graph for each allocation site. These hidden class graphs are output as the result of profiling.

In addition to the graphs, the profiling VM collects information to identify hot hidden classes. We define the *hotness* of a hidden class by the maximum number of objects simultaneously alive that have the hidden class during a GC. At every GC in the profiling run, we count, for each hidden class, the objects with this hidden class and update its hotness.

As we mentioned in Section 2, hidden classes hold pointers to prototype objects to enable inline caching for prototype properties. However, prototype objects are first-class objects created dynamically. In the profiling run, we do not collect the contents of prototype objects, but collect their addresses as identifiers of prototypes. They are only used to judge whether two hidden classes have the same prototype or not.

### 4.2 Hidden Class Optimization

Based on the profiled graph and allocation sites, we construct an optimized hidden class graph as follows:

1. For each input hidden class graph, move branches to optimize to the hot transition chains.
2. For each hidden class graph, eliminate cold intermediate hidden classes.
3. Merge the hidden class graphs of allocation sites with identical hidden class graphs.

***Moving Branches.*** The optimizer moves branches to linearize the paths to hot hidden classes. It searches for the hidden classes that have two or more transitions from the root. When it finds such a hidden class, it computes the weight of each transition—the sum of the hotness values of the hidden classes in the subtree connected by the transitions (the hidden class graph is a tree). If the vast majority (e.g. 80%) of the weight is held by one transition, the step moves the other transitions down the tree, leaving the major transition, as shown in Figure 11.

***Elimination of Cold Intermediate Hidden Classes.*** The optimizer eliminates hidden classes if they are cold and intermediate ones. It preserves hot hidden classes as explained in Section 3. It also preserves hidden classes that have multiple transitions. Because it previously moved the minor branches, the remaining ones are worth preserving.

***Merging Graphs.*** The hidden class graphs for different allocation sites may be identical, that is, they may have the same transitions and the same prototype. For example, if object literals with the same initial properties occur in multiple places in the program, and if objects created there do not have any additional property, the hidden class graphs for the object literal allocation sites become identical. For allocation sites with identical hidden class graphs, the optimizer makes sure they share the same hidden class graph.

### 4.3 Execution with Optimized Hidden Class Graph

After all optimizations are applied, the VM can load the optimized hidden class graph when executing the corresponding program. It installs the entry points of the graph at the allocation sites. When an allocation site is executed, the VM creates an object directly with the optimized hidden class.

For hidden classes with properties, the VM allocates space for the properties directly *in* the object instead of in a separate property array. The properties are initialized with `JS_EMPTY` We use `JS_EMPTY` to distinguish properties with `undefined` values from properties that an object does not (yet) have. Where an object transitions to another hidden class with additional properties, a property array is allocated and installed in the last slot used for the in-object properties, with the last in-object property moved into the property array. Thus the property array has one more slot than the

number of added properties. If the object transitions further, the property array is re-allocated to add space for new properties.

***Filling The Prototype Object Field.*** As mentioned in Section 2, eJSVM stores the pointer to the prototype in the hidden class. However, the loaded hidden classes do not hold pointers to prototype objects as these are created dynamically. Instead, the prototype pointer is installed when the allocation site is executed for the first time. When an allocation site is executed, the prototype object for the new object is provided. If the installed hidden class does not yet have a pointer to a prototype object, the runtime installs the pointer in the prototype object in all hidden classes in the hidden class graph installed at the allocation site.

We do this assuming that the prototype object for the objects allocated at the same allocation site remains the same across executions, i.e., in the profiling run and the actual run. However, when this assumption does not hold, or the program explicitly calls `setPrototypeOf`, the object departs from the loaded graph and a new hidden class is created, as if for unoptimized execution. This is needed when the prototype object is different because they object may behave different to any transitions in the loaded hidden class graph. This also means that the profiling was insufficient.

### 4.4 Prototype Polymorphic Allocation Sites

Because hidden classes have pointers to prototypes, new hidden classes must be created for objects with different prototypes. However, a program may create objects with different prototypes at the same allocation site. We call such an allocation site *prototype-polymorphic*.

We currently do not optimize prototype-polymorphic allocation sites. This is because, in the actual run, we have no way to select the right hidden class from multiple hidden classes associated with a prototype-polymorphic allocation site, since the hidden classes loaded at the actual run do not yet have prototypes.

## 5 Evaluation

We evaluated the proposed offline hidden class optimization using the Are We Fast Yet benchmarks [13]. In addition, we created variants of the CD and Json benchmarks, in which objects behave more dynamically than the original ones.

CD simulates an aircraft collision detector and relies heavily on a red-black tree. Because it is ported from Java [11], it initializes all properties of tree node objects eagerly with `null`. This includes the `left` and `right` properties, which hold child nodes, if any. As a result, properties may take up space unnecessarily. We created a modified CD-dyn which does not initialize properties of node objects eagerly, more in keeping with common JavaScript style.

The Json benchmark parses JSON input and constructs an object tree representing the JSON objects with objects

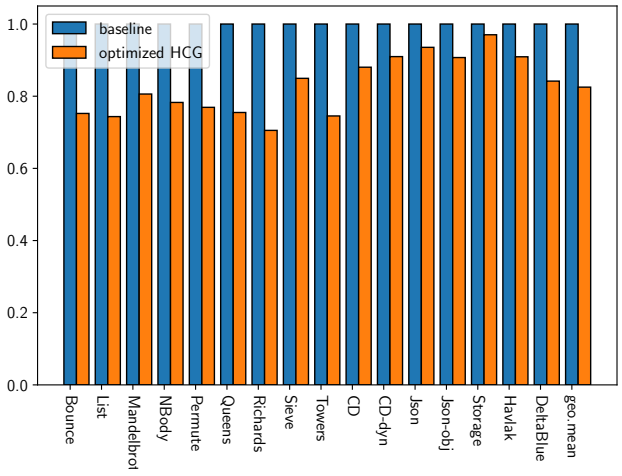


**Figure 12.** Memory footprint normalized to the original eJSVM. It is reduced by 3–30% across all benchmarks.

that use for instance two arrays, one for the property names and one for the values, instead of using object properties directly. As a consequence, the parsed objects have the same hidden class. Again, since the benchmark originates from a JSON parser written in Java, it does not utilize JavaScript's objects fully. We created the Json-obj variant to use object properties directly so that the parsed objects have a variety of hidden classes.

We ran each benchmark five times on a workstation with a Xeon W-2235 processor whose clock frequency was fixed to 3.80GHz by turning off turbo boost and hyperthreading, and using the performance governor. We used the Ubuntu Linux 20.04.5 TLS operating system and GCC 9.4.0 compiler to compile eJSVM. eJSVM was configured for a 64-bit environment and uses the Fusuma sliding compaction GC [14].

### 5.1 Memory Footprint

Since our primary goal is to reduce memory footprint, we start by assessing memory usage. We measure the total volume of live objects periodically, and used the largest volume as an approximation of the memory footprint. We modified eJSVM so that a GC happens after every 10KB of allocation to measure memory usage.

Figure 12 shows the memory footprint normalized to the original eJSVM for each program. Our offline hidden class graph optimizations reduced memory footprint for all programs. Part of the reduction comes from the reduced memory footprint for the hidden classes themselves. As shown in Figure 13, this was reduced by more than 50% for all programs, and by 61.9% on average.

Hidden classes are similar to class objects in other languages, and thus, their number tends to be stable regardless of the number of normal objects used in the program. Therefore, the memory footprint reduction ratio tends to

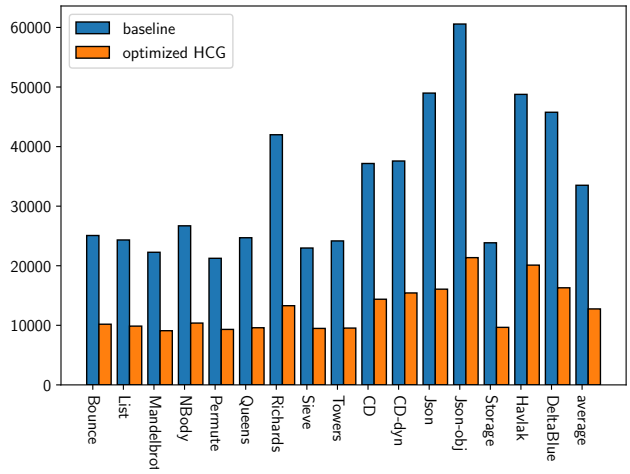


**Figure 13.** Memory used for hidden-class-related data. Averages of the numbers collected after all GC cycles. All benchmarks see at least a 50% reduction, with 61.9% on average.

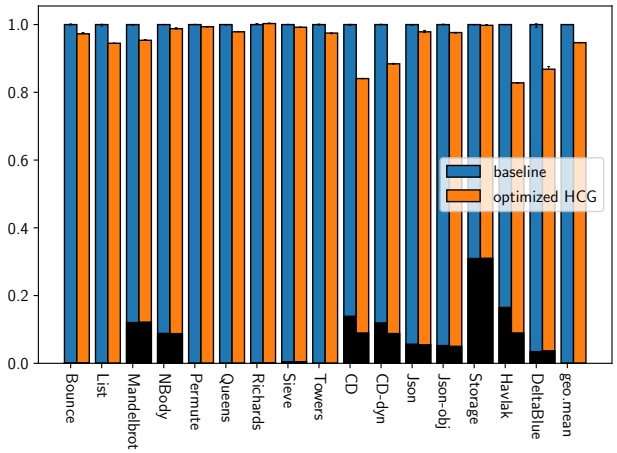


**Figure 14.** Execution time normalized to the original VM. Black parts indicate GC times. Execution speeds improve by 6.3% on average. Only Richards sees a 0.3% slowdown. Error bars indicate quartiles.

be large for programs with small footprints. For example, Richards uses only few objects and the footprint was reduced by 28KB, from 95KB to 67KB (29.5%). For Storage, which allocates many arrays, the footprint was reduced by 17KB, from 583KB to 566KB (3%).

CD, CD-dyn, Json, and Json-obj had similar memory footprints to Storage. DeltaBlue had a much larger one with 7.5MB for the original VM. However, these benchmarks exhibit a larger footprint reduction than Storage. This indicates that normal objects have became smaller as pretransitioning allows properties to be allocated in-object, eliminating the memory cost of an external property array. In eJSVM, this saves two words per object: the object's property pointer field (which can be repurposed for an in-object property) and the external property array's header.

### 5.2 Execution Speed

Figure 14 shows the elapsed times for each benchmark program normalized to the original VM. Black bars indicate time spent in stop-the-world GC. As shown in the figure, execution speed improved by around 15% (11.6–17.6%) for some benchmarks and did not change significantly for the others. We emphasize that it did not degrade for any program, except Richards, for which the execution time was 0.3% slower than the original VM. Overall, execution speeds improved by 6.3% on average.

Eliminating intermediate hidden classes can improve execution time because it reduces reallocation of the property arrays at each hidden class transition. The reduction in GC time for CD, CD-dyn, and Havlak may be cause by a reduction of property array reallocations.

For DeltaBlue, execution time reduced by 13.2% while GC time did not. To investigate the reason, we measured the miss ratios of inline caches. As Figure 15 shows, VMs with optimized hidden class graphs exhibited lower cache miss ratios for most of the programs that had noticeable miss ratios in the original VM. For CD, Havlak, and DeltaBlue, around 3.1–8.1% of property accesses missed inline caches for the original VM, and almost all accesses hit inline caches for the VM with optimized hidden class graphs.

For CD-dyn, even with the optimized hidden class graph, some inline cache misses remained. This is because there were five variations of hidden classes for the node objects of the red-black tree. The node objects were created without properties for child nodes, `left` or `right`. This hidden class was hot because some of them remained leaf nodes until the end. They became internal nodes when a property `left` or `right` was added. The hidden classes with one of `left` or `right` were also hot. Some objects then obtained the other property, transitioning to the final hidden classes. Note that the final hidden classes could not be combined because `left` and `right` were added in different order, and hence have different offsets.

Mandelbrot has very few property accesses, only 232 in total. Most property accesses were the first access at the site, and hence needed to initialize the inline caches. For comparison, CD has more than one million accesses.

## 6 Related Work

***Reusing Profiling Information.*** Using profiling information for optimization decisions across runs of a VM has been studied for Java and JavaScript JIT compilation. For instance, Arnold et al. [1] kept profiling information collected in a persistent repository. They computed the optimization level to be applied to each method in the next run from

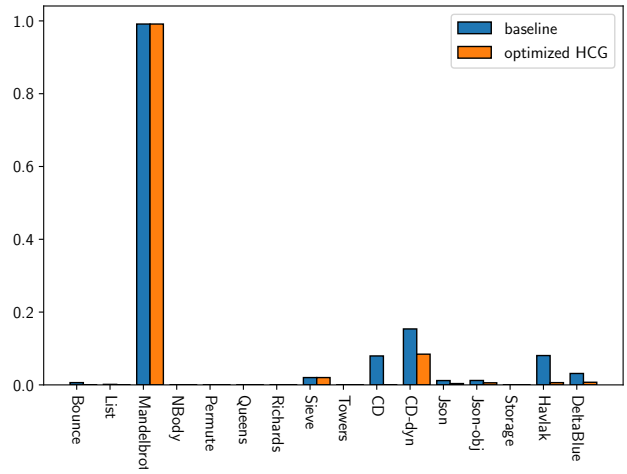


**Figure 15.** Inline cache miss ratio. They remain the same or are reduced by 3.1–8.1%.

the collected profiling information. Majo et al. [12] also proposed to reuse profiling information collected in previous runs for the JIT compilations in following runs. They mainly focused on the reduction of time to the peak performance, i.e. warmup time. Park et al. [16] even keep the compiled code from previous runs and solve issues for code relocation and validating speculations.

In contrast, we profile the hidden classes graph and optimize it to improve performance in terms of memory use and execution time.

Choi et al. [5] reuse inline caches in the V8 JavaScript VM. Inline caches depend on the hidden classes. Although the hidden classes are created dynamically, they assumed static behaviour in practical programs, i.e., that property access sites see the same hidden classes across different runs. With this assumption, they group the property access sites by hidden classes. When one of the access sites in a group is executed, they constructed inline caches for all access sites in the group. In our proposal, hidden classes are also static but prototype objects are dynamically created. Thus, we install prototype objects when an allocation site is executed for the first time.

***Offline Optimization.*** Ottoni and Liu [15] optimize objects' layout offline using profiling information. They apply it to Hack and PHP, which are executed as dynamic languages. They reorder the fields of structures to improve access locality. They collect property access counts in the profiling run, and arrange fields in the order of their access counts. Their workflow is similar to ours, but our target is the hidden class graph. In addition, we focus on reducing memory footprint.

***Hidden Classes and Inline Caching.*** Wu et al. [19] leverage the class hierarchy provided by the programmer in TypeScript and create subtype polymorphic inline caching. We improve inline cache hit ratios by reducing the variation of the hidden classes, eliminating cold hidden classes.

While the original idea of maps, or hidden classes as they are called today, originated in Self [3], others such as Wöß et al. [20] refined the ideas, for instance by constructing the mapping information lazily by reconstructing it on the first access from the transitions in the hidden class graph. Since we optimize offline, we can eliminate intermediate hidden classes, which gives a similar gain with the added benefit of avoiding the intermediate transitions.

Another aspect of the work by Wöß et al. [20] is that it encodes type information in hidden classes. Cheng et al. [4] take a similar approach. For eJSVM, we currently do not capture types in hidden classes.

In addition to optimizations for objects and their hidden classes, it could be beneficial to optimize arrays and other builtin collections [2, 6, 9]. Storage strategies benefit from communicating usage details to the allocation sites, for which we could use our approach in the future.

## 7 Conclusion and Future Work

We presented the offline optimization of hidden class graphs for JavaScript VMs targeting memory constrained embedded systems. Based on a closed-world assumption, we harvested the hidden class graph and the usage of each hidden class from the profiling run, and constructed the optimized hidden class graph for the actual run. The primary goal of the optimization is to reduce memory footprint. We reduced it by eliminating most intermediate hidden classes and placing properties *in* objects, thus eliminating property arrays. As a result, we reduced the memory used by the hidden class graph by 61.9% on average and overall footprint by 3–30%. Execution speed was also improved by reducing reallocation of the property arrays and reducing the polymorphism of the inline caches. The improvements were 11.6–17.6% in the best cases.

One of our future goals is to place the optimized hidden class graph in read-only flash memory rather than loading them into SRAM on actual execution. Employing techniques to reduce memory footprint, such as storage strategies, is another goal for the future.

## Acknowledgments

We are grateful for the support of the JSPS through KAKENHI grant number JP18KK0315. This work was also supported by the Engineering and Physical Sciences Research Council (EP/V007165/1) and a Royal Society Industry Fellowship (INF\R1\211001).

## References

[1] Matthew Arnold, Adam Welc, and V. T. Rajan. 2005. Improving virtual machine performance using a cross-run profile repository. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented*

*Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 297–311. https://doi.org/10.1145/1094811.1094835

[2] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. ACM, 167–182. https://doi.org/10.1145/2509136.2509531

[3] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 49–70. https://doi.org/10.1145/74878.74884

[4] Lin Cheng, Berkin Ilbeyi, Carl Friedrich Bolz-Tereick, and Christopher Batten. 2020. Type Freezing: Exploiting Attribute Type Monomorphism in Tracing JIT Compilers. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO'20)*. ACM. https://doi.org/10.1145/3368826.3377907

[5] Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable inline caching for JavaScript performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 889–901. https://doi.org/10.1145/3314221.3314587

[6] Daniel Clifford, Hannes Payer, Michael Stanton, and Ben L. Titzer. 2015. Memento mori: dynamic allocation-site-based optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management, ISMM 2015, Portland, OR, USA, June 13-14, 2015*, Antony L. Hosking and Michael D. Bond (Eds.). ACM, 105–117. https://doi.org/10.1145/2754169.2754181

[7] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (Salt Lake City, Utah, United States). ACM, 297–302. https://doi.org/10.1145/800017.800542

[8] Kai Grunert. 2020. Overview of JavaScript Engines for Resource-Constrained Microcontrollers. In *2020 5th International Conference on Smart and Sustainable Technologies (SpliTech)*. 1–7. https://doi.org/10.23919/SpliTech49282.2020.9243749

[9] Johannes Henning, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. 2020. Toward presizing and pretransitioning strategies for GraalPython. In *Programming'20: 4th International Conference on the Art, Science, and Engineering of Programming, Porto, Portugal, March 23-26, 2020*, Ademar Aguiar, Shigeru Chiba, and Elisa Gonzalez Boix (Eds.). ACM, 41–45. https://doi.org/10.1145/3397537.3397564

[10] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: European Conference on Object-Oriented Programming (LNCS, Vol. 512)*. Springer, 21–38. https://doi.org/10.1007/BFb0057013

[11] Tomas Kalibera, Jeff Hagelberg, Petr Maj, Filip Pizlo, Ben Titzer, and Jan Vitek. 2011. A family of real-time Java benchmarks. *Concurrency and Computation: Practice and Experience* 23, 14 (2011), 1679–1700. https://doi.org/10.1002/cpe.1677

[12] Zoltan Majo, Tobias Hartmann, Marcel Mohler, and Thomas R. Gross. 2017. Integrating Profile Caching into the HotSpot Multi-Tier Compilation System. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes, ManLang 2017, Prague, Czech Republic, September 27 - 29, 2017*. ACM, 105–118. https://doi.org/10.1145/3132190.3132210

[13] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) *(DLS'16)*. ACM, 120–131. https://doi.org/10.1145/2989225.2989232

[14] Hiro Onozawa, Tomoharu Ugawa, and Hideya Iwasaki. 2021. Fusuma: double-ended threaded compaction. In *ISMM '21: 2021 ACM SIGPLAN International Symposium on Memory Management, Virtual Event, Canada, 22 June 2021*, Zhenlin Wang and Tobias Wrigstad (Eds.). ACM, 94–106. https://doi.org/10.1145/3459898.3463903

[15] Guilherme Ottoni and Bin Liu. 2021. HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, 340–350. https://doi.org/10.1109/CGO51591.2021.9370314

[16] Hyukwoo Park, Sungkook Kim, Jung-Geun Park, and Soo-Mook Moon. 2019. Reusing the Optimized Code for JavaScript Ahead-of-Time Compilation. *ACM Transactions on Architecture and Code Optimization* 15, 4 (Jan. 2019), 1–20. https://doi.org/10.1145/3291056

[17] Tomoharu Ugawa, Hideya Iwasaki, and Takafumi Kataoka. 2019. eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems. *Journal of Computer Languages* 51 (2019), 261–279. https://doi.org/10.1016/j.cola.2019.01.003

[18] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 1–29. https://doi.org/10.1145/3360610

[19] Zhefeng Wu, Zhe Sun, Kai Gong, Lingyun Chen, Bin Liao, and Yihua Jin. 2020. Hidden Inheritance: An Inline Caching Design for TypeScript Performance. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 1–29. https://doi.org/10.1145/3428242

[20] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 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* (Cracow, Poland) *(PPPJ '14)*. ACM, 133–144. https://doi.org/10.1145/2647508.2647517

Received 2022-09-18; accepted 2022-10-05