Reusable Inline Caching for JavaScript Performance

Jiho Choi
University of Illinois at Urbana-Champaign
USA
jchoi42@illinois.edu

Thomas Shull
University of Illinois at Urbana-Champaign
USA
shull1@illinois.edu

Josep Torrellas
University of Illinois at Urbana-Champaign
USA
torrella@illinois.edu

Abstract

JavaScript performance is paramount to a user's browsing experience. Browser vendors have gone to great lengths to improve JavaScript's steady-state performance. This has led to sophisticated web applications. However, as users increasingly expect instantaneous page load times, another important goal for JavaScript engines is to attain minimal startup times.

In this paper, we reduce the startup time of JavaScript programs by enhancing the reuse of compilation and optimization information across different executions. Specifically, we propose a new scheme to increase the startup performance of Inline Caching (IC), a key optimization for dynamic type systems. The idea is to represent a substantial portion of the IC information in an execution in a context-independent way, and reuse it in subsequent executions. We call our enhanced IC design *Reusable Inline Caching (RIC)*. We integrate RIC into the state-of-the-art Google V8 JavaScript engine and measure its impact on the initialization time of popular JavaScript libraries. By recycling IC information collected from a previous execution, RIC reduces the average initialization time per library by 17%.

CCS Concepts • **Software and its engineering → Just-in-time compilers**; **Scripting languages**; *Polymorphism*; *Classes and objects.*

Keywords Inline Caching, JavaScript, Scripting Language, Dynamic Typing

ACM Reference Format:
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 '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3314221.3314587

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314587

1 Introduction

In recent years, JavaScript has become a very popular programming language. Because JavaScript is the only programming language supported by all browsers, it is widely used for web development, and is known as the de facto programming language of the web.

Achieving efficient execution of JavaScript is a challenging task. A primary reason for this is the dynamic nature of JavaScript. JavaScript is a dynamically-typed programming language which supports dynamic properties. This means that properties can be added and removed from objects dynamically throughout execution. In addition, since JavaScript is a scripting language, code is compiled dynamically using Just-In-Time (JIT) compilation. Despite all of these difficulties, JavaScript's importance means that vast industrial resources have been spent to ensure that JavaScript performs well in browser implementations.

JavaScript's performance improvements have coincided with the development of advanced, immersive web applications. Better JavaScript performance has enabled increasingly complicated applications to be web-based, such as office productivity suites, map services, and interactive games. However, another important metric when evaluating web applications is their page load time. According to a series of user surveys [3, 19, 28, 33], while the majority of users waited up to eight seconds for a page load in 1999, acceptable wait time shrunk to two seconds by 2014. Furthermore, the industry is pushing the bar even higher, by advocating a sub-second wait time so as not to interrupt a user's flow of thoughts [23].

Reducing wait times is complicated by the fact that websites are continuing to grow in size and complexity. Figure 1 shows these conflicting trends. As shown in the figure, the average number of JavaScript requests in the top 1000 websites has gone up from 12 in 2010 to 28 in 2015. Considering that the initialization of each JavaScript library takes tens to hundreds of milliseconds [24], it will be challenging to meet the ever increasing user expectations.

The goal of this paper is to reduce user wait times by improving JavaScript's startup performance. Current JavaScript implementations use online profiling techniques to improve steady-state performance. However, they do not attempt to reuse this profiling information across different executions. There are some techniques that reuse compilation and optimization information across different executions in

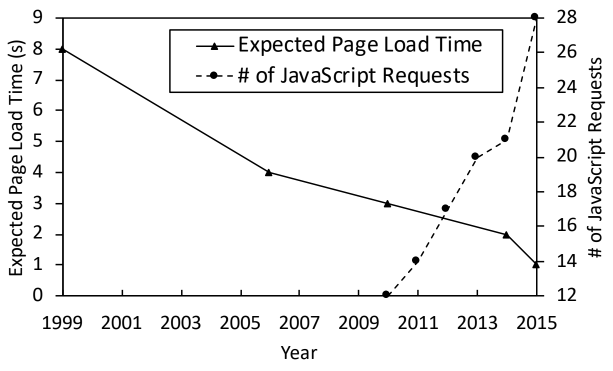


Figure 1. Conflicting trends of user expectation for page load time and website complexity.

PHP [1, 25] and Java [21, 31]. However, such techniques are not directly applicable to or are unsuitable for JavaScript due to JavaScript's highly dynamic nature.

To enable program information reuse across executions, we focus on Inline Caching (IC) [11] — a technique used in dynamically-typed languages to specialize sites in the program that access objects. The IC structures are dynamically built with profile information gathered during execution, and they help improve performance. Unfortunately, in current JavaScript implementations, such structures are cleared and repopulated at each execution. This is because they contain context-dependent information, such as the memory addresses of heap objects, which are not consistent across runs.

In our analysis of IC in JavaScript, we find that, while IC structures have context-dependent information, many facets of their internals are in fact context-independent. In particular, we make two observations. First, many of the code routines invoked by the IC are context-independent and can be reused across executions. Second, sets of program sites that access the same program objects, often update the IC structures in similar ways. This property, which is retained across executions, allows us to expedite the process of repopulating IC structures in subsequent executions.

Based on these insights, in this paper we propose a new IC design called *Reusable Inline Caching (RIC)*, which enables information reuse across executions. RIC extracts the context-independent portion of the IC information from the initial execution, and reuses it in subsequent executions to significantly reduce JavaScript startup time.

We integrate RIC into the state of the art Google V8 JavaScript compiler [8] and measure its impact on the initialization time of popular JavaScript libraries. By recycling the IC information collected from a previous execution, RIC improves the initialization of libraries: it reduces the average dynamic instruction count by 15%, and the average execution time by 17%. The contributions of this paper are as follows:

- Characterize the IC overheads of popular JavaScript libraries during initialization.
- Identify opportunities to reuse IC information across executions.
- Propose RIC, a new IC design to enable the reuse of IC information across executions.
- Implement RIC in the state of the art Google V8 JavaScript compiler.
- Provide a detailed evaluation of RIC's performance.

2 Background

2.1 JavaScript Execution

As JavaScript was initially designed to be embedded within browsers, JavaScript implementations commonly provide APIs to control execution. The host system in which a JavaScript runtime is embedded, such as a web browser or Node.js, is responsible for scheduling JavaScript execution. Normally, the host system can drive JavaScript execution through two means: loading new scripts (i.e., *initialization*) and adding new events (i.e., *event handling*).

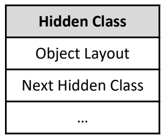
Initialization occurs when the host system first loads a JavaScript source file into the JavaScript runtime. For example, when a browser encounters a script element during HTML parsing, it passes the JavaScript source code in the tag to the JavaScript runtime. Alternatively, the host system can register JavaScript functions to handle events, such as user input. These functions are then executed when the corresponding events are triggered.

In this paper, we focus on JavaScript initialization, as it directly affects page load performance. The page cannot be fully loaded until all included JavaScript source code has initialized. For example, the initialization can block DOM object construction and page rendering. In addition, user interaction is typically disabled until initialization is complete.

2.2 Hidden Classes

JavaScript is a dynamically-typed programming language, where objects can have properties added to, and deleted from at runtime. Such dynamism prevents JavaScript compilers from constructing a fixed object layout before execution. However, to generate efficient code, it is crucial for the compiler to have some notion of object type. To resolve this conflict, JavaScript implementations dynamically create *Hidden Classes* for objects. This concept was first introduced in Self [10]. The basic idea is to assign each object a hidden class, which contains information about the current layout of the object. Objects created in the same way are assigned the same hidden class. Grouping objects in this manner helps to enable optimizations. Throughout this paper, we will use the terms "type" and "hidden class" interchangeably.

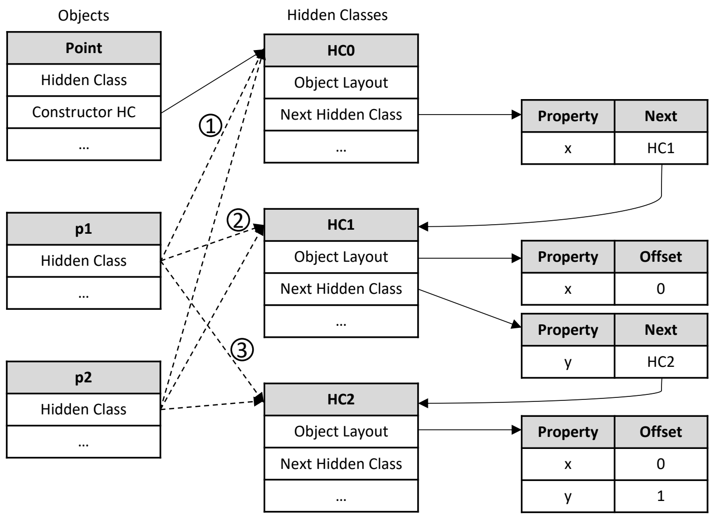
Figure 2(a) shows the general structure of a hidden class used by V8. The *Object Layout* field points to a table that keeps the object layout, with a list of (property, offset) pairs.



(a) Hidden class structure.

```
1: function Point(x, y) {
2:  this.x = x;
3:  this.y = y;
4: }
5: p1 = new Point(10, 20);
6: p2 = new Point(30, 40);
```

(b) Example JavaScript code.



(c) Hidden class transition example.

Figure 2. Hidden class structure and example of hidden class transition.

The *Next Hidden Class* field points to a table that keeps a list of (property, hidden class) pairs. The table tells the next hidden class to transition to, when the new property is added.

To understand the operation of objects and hidden classes, consider the simple code example of Figure 2(b). The code declares function `Point`, which sets coordinates `x` and `y` of a point. Then, the code creates and sets points `p1` and `p2`.

When a function is declared, as in Line 1, the runtime allocates an object for the function (*Point*, at the top left of Figure 2(c)) and a hidden class for the object to be constructed by the function (*HC0*, at the top center of Figure 2(c)). The *Constructor Hidden Class (HC)* of the function object points to the hidden class, whose fields are initially empty.

In Line 5, when the code calls the `Point` function to create `p1`, the runtime creates an object (`p1`, at the center left of Figure 2(c)). The object initially points to hidden class `HC0` (①). As the function `Point` executes and adds properties to the point in Lines 2 and 3, new hidden classes are created, and the *Hidden Class* field of object `p1` changes (② and ③).

Specifically, in Line 2, a new property `x` is added to `p1`. At this point, the runtime creates a new hidden class `HC1`, with an *Object Layout* that has property `x` at offset 0 (center right of Figure 2(c)). At the same time, `HC0`'s *Next Hidden Class* is set to point to `HC1` (top right of Figure 2(c)), and `p1`'s *Hidden Class* is set to point to `HC1` (②).

Similarly, when the new property `y` is added to `p1` in Line 3, the runtime creates a new hidden class `HC2`, with an *Object Layout* with `x` at offset 0 and `y` at offset 1 (lower right of Figure 2(c)). At the same time, `HC1`'s *Next Hidden Class* and `p1`'s *Hidden Class* pointers are set to point to `HC2` (③).

These hidden classes are created only for a new transition. When point `p2` is created in Line 6, only object `p2` is allocated (lower left of Figure 2(c)). As execution proceeds, this object's *Hidden Class* pointer will successively point to hidden classes `HC0`, `HC1`, and `HC2`.

2.3 Inline Caching

One of the fundamental optimization techniques enabled by hidden classes is *Inline Caching* (IC) [11]. To understand IC, we call *Object Access Site* any location in the program where an object property is read or written. IC is based on the empirical evidence that the objects accessed at a particular object access site often have the same hidden class or classes.

Without IC, the runtime system would be invoked at every object access site and, after identifying the type of the incoming object, would perform the appropriate load or store operation at the appropriate offset. The *incoming object* is the object whose property is being read or written at the site. Unfortunately, this operation has substantial overhead.

The idea behind IC is that, at each site, every time that the runtime encounters a new hidden class for the site, it generates a handler routine with the operation that needs to be performed for that hidden class. Then, the runtime specializes the code at the site so that it checks the hidden class of each incoming object and, if the hidden class has been seen before, it calls the corresponding handler. Hence, when a site encounters a hidden class seen before, the execution is highly efficient.

V8 uses an out-of-line approach to IC. Instead of directly specializing the machine code, it creates a per-function data structure called `ICVector` (Figure 3). For each object access site in the function, the `ICVector` contains one or more slots. Each slot corresponds to one different hidden class encountered at this site in the past. A slot contains a tuple ($\texttt{HC}_{Addr}$, `Handler`). $\texttt{HC}_{Addr}$ is a pointer to the hidden class; `Handler` is a pointer to the handler routine that performs the operation for the hidden class. For example, Figure 3 shows an `ICVector` with three access sites, each with two slots.

Site	Object Access Site 1	Object Access Site 2	Object Access Site 3
HC_{Addr} Handler	$\binom{HC_{11}}{Handler_{11}}\binom{HC_{12}}{Handler_{12}}$	$\binom{HC_{21}}{Handler_{21}}\binom{HC_{22}}{Handler_{22}}$	$\binom{HC_{31}}{Handler_{31}}\binom{HC_{32}}{Handler_{32}}$

Figure 3. `ICVector` data structure.

The information stored in the `ICVector` helps speed up accesses to objects at these sites. If the incoming object's hidden class matches one of the HC_{Addr} for the site, execution directly transfers to the handler routine stored in `Handler`, without having to call the runtime. Otherwise, an *IC Miss* has occurred. At this point, the runtime is invoked to perform the load or store, and to augment the `ICVector` with an additional slot for the site.

At the start of execution, the `ICVector` is empty. As execution progresses, the `ICVector` is filled with (HC_{Addr}, `Handler`) pairs for each object access site. An object access site that only encounters objects of a single hidden class is called *monomorphic*; if it encounters objects of multiple hidden classes, it is called *polymorphic*.

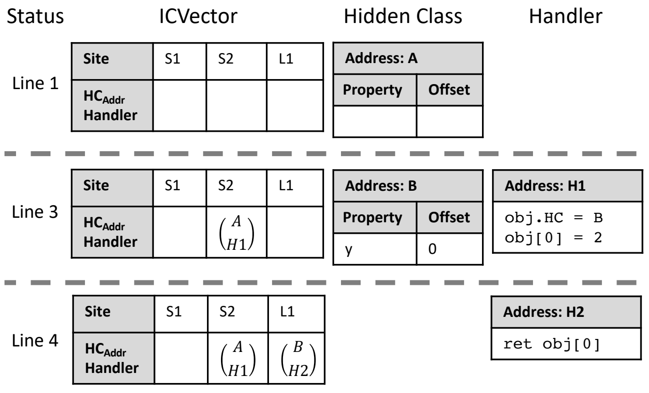
2.4 Putting It All Together

Based on the previous discussion, we now show an example of how the IC information is generated during execution.

Figure 4(a) shows a JavaScript source code example that creates an empty object `o` (Line 1), adds property `x` if a branch is taken, adds property `y`, and finally prints out the value of property `y`. There are three object access sites in the example. Those in Lines 2 and 3 add a property and store a value to it. We call them sites `S1` and `S2`, respectively. The one in Line 4 loads the value of a property. We call it site `L1`.

```
1: var o = {};
2: if (…) o.x = 1; // S1
3: o.y = 2; // S2
4: print(o.y); // L1
```

(a) Example JavaScript code.



(b) Structures created by V8.

Figure 4. Example of V8 structures.

Figure 4(b) shows the `ICVector` structure for the code, as well as a simplified representation for the hidden classes and handler routines that V8 creates as the code executes. The `ICVector` has a column for each of the object access sites in the code (`S1`, `S2`, and `L1`). Figure 4(b) shows the state of the data structures after a given line of code from Figure 4(a) is executed. Specifically, after Line 1 is executed, the `ICVector` is empty, and there is an empty hidden class for object `o`. Let us call this hidden class `A`, as it is allocated at memory address `A`.

Let us now assume that the branch on Line 2 is not taken, and `S1` is not accessed. As Line 3 executes, `ICVector`'s column for `S2` is checked to see if it has information that can be used to avoid a runtime call. However, at this point, this column is empty. Therefore, an IC miss occurs and the runtime is invoked. The runtime creates: (i) the appropriate handler routine and (ii) a new hidden class, since the hidden class `A` of the incoming object does not contain information about property `y`. This is shown in Line 3 of Figure 4(b). Let us assume that the handler is located at address `H1`, and the new hidden class at address `B`. The property `y` is placed at offset `0`. Next, the runtime adds a slot to the column corresponding to this object access site with the addresses of the hidden class `A` of the incoming object and the handler `H1`. Finally, it invokes the handler, which fills the object with its hidden class and the value 2 at offset `0`. From now on, if this site is used again and the incoming hidden class is `A`, the runtime will not be called. Throughout the remainder of the paper, we call object access sites that create new hidden classes *transitioning* object access sites.

The execution of Line 4 also causes an IC miss, since ICVector's `L1` column is empty. As before, the runtime is called to create a new handler. Let us assume that this handler is placed at address `H2`. Since the incoming object's hidden class `B` already contains property `y`, the runtime does not create a new hidden class. It simply adds a slot in column L1 of the ICVector with the addresses of the incoming object's hidden class `B` and the new handler `H2`, and invokes the handler. The handler returns the value of `y` at offset `0`.

If sites S2 and L1 are later accessed by an object with a different hidden class than in the example, another slot is added to the corresponding column.

3 Characterizing Inline Caching

While IC improves the performance of JavaScript programs, IC misses still induce significant overhead during the initial sections of programs. Importantly, this overhead does not disappear as a program is re-executed. Indeed, V8 discards the `ICVector` data at the end of every execution, and recreates it from scratch at the beginning of each new execution. In this section, we estimate the overhead of IC misses, explain why IC state is difficult to reuse across executions, and provide some motivation for attempting to reuse it.

3.1 Overhead of IC Miss Handling

To estimate the overhead of IC misses, we take a set of seven popular JavaScript libraries and profile their initialization. Specifically, we count the number of instructions that execute during IC misses. Such instructions, which are executed by the runtime, look up the incoming object's layout to find the property requested by the object access site, generate a specialized handler routine, and update the `ICVector` for future accesses to this site.

Figure 5 breaks down the instructions executed by the libraries during initialization into those used to handle IC misses and the rest of instructions. The latter include JavaScript code execution and the rest of runtime. The libraries used are discussed in Section 6. The figure shows that IC miss handling accounts for a substantial fraction of the instructions executed during initialization in all libraries. On average, it accounts for 36% of the instructions.

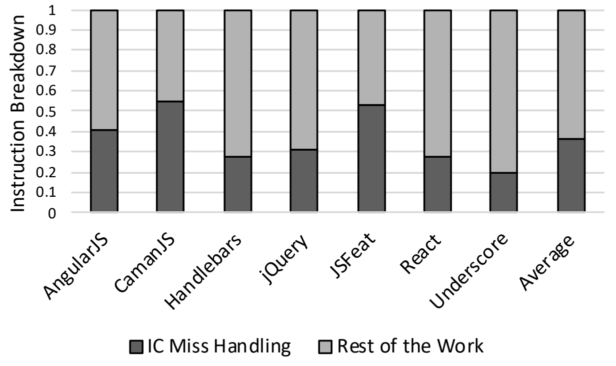


Figure 5. Instruction breakdown during the initialization of JavaScript libraries.

3.2 Context-Dependence of IC

The previous data suggests that if one could reuse IC state across program executions such that IC misses were avoided, the initialization of JavaScript programs would likely speed up substantially.

However, only context-independent state can typically be reused across executions. We say that information is context-independent if it is not tied to any memory addresses other than the addresses of *Built-in* objects. Built-in objects are defined by the language standard, and include fundamental objects (e.g., `Object` and `Function`) and utility functions (e.g., `Math` and `String`) [12].

Unfortunately, hidden classes are context-dependent. Figure 2(a) shows the structure of a hidden class used by V8. While the *Object Layout* field is context-independent, the rest of the fields are context-dependent. For example, the *Next Hidden Class* field points to various hidden classes that are created as new properties are added. The values of such pointers vary across executions. Moreover, one of the fields not shown in Figure 2(a) is the *Prototype* field, which points to the prototype object. Note that JavaScript uses prototyping to emulate the class inheritance of statically-typed languages such as C++ and Java. Every JavaScript function has a prototype object, and all the objects created by the same function inherit from the function's prototype object. Since prototype objects are dynamically allocated on the heap, the *Prototype* field of hidden classes is also context-dependent.

In addition, some handlers are context-dependent and, therefore, cannot be reused across executions. For example, when accessing an inherited property, the handler traverses the chain of prototype objects to locate the property. It compares the hidden classes of the prototype objects of the incoming object to those embedded in the handler code. The result is context-dependent state. Another common example is a handler adding a new property to an object (e.g., handler `H1` in Figure 4(b)). The addition of a new property triggers the hidden class transition of the incoming object. The handler embeds the hidden class to which the incoming object transitions (e.g., `B` in Figure 4(b)). Such handlers are context-dependent, since they embed hidden class information.

On the other hand, there are some context-independent handlers. One example is handler `H2` in Figure 4(b). The handler simply accesses a property at a fixed offset within the incoming object.

3.3 Opportunities for Reuse

Despite these difficulties, there are opportunities for reuse. For example, the interaction with many websites starts by initializing a set of popular JavaScript libraries. Their execution is fairly deterministic and context-independent. Some of the IC state created by initializing these libraries could be reused across executions.

To investigate the potential of this idea, we instrument the libraries of Section 3.1 to collect various statistics related to IC. Table 1 shows the results. The first column shows the number of different hidden classes encountered during the initialization of the libraries. The second column shows the number of IC misses. In all the libraries, the number of IC misses is much higher than the number of different hidden

Table 1. Statistics related to IC use during the initialization of JavaScript libraries.

Library	# of Diff. Hidden Classes	# of IC Misses	# of IC Misses per HC	% of Context Independent Handlers
AngularJS	138	799	5.8	62.5
CamanJS	99	383	3.9	61.8
Handlebars	88	541	6.2	63.2
jQuery	271	1547	5.7	57.3
JSFeat	116	323	2.8	51.7
React	360	2356	6.5	82.3
Underscore	123	295	2.4	38.1
Average	171	892	4.8	59.6

Table 2. Main ideas in Reusable Inline Caching (RIC).

Linking of object access sites	An IC miss in the *Triggering* site causes all of its *Dependent* sites to populate their ICs.
Reusing handlers	Context-independent handlers from the *Initial* run are reused in the *Reuse* run.
Hidden class validation	Incrementally certify that hidden classes in the *Reuse* run match those in the *Initial* run.

classes. The third column shows the ratio of the first two columns. Since a given hidden class does not miss more than once in a given object access site, this means that the same hidden class is encountered in several different object access sites. On average, a hidden class is encountered in about 5 object access sites. The last column shows the fraction of handlers that are context-independent and reusable across executions. On average, about 60% of the handlers generated in the libraries are context-independent.

Overall, this characterization shows that there may be opportunities to minimize the overhead of IC misses by reusing IC information across executions. First, the fact that the same hidden class causes misses in multiple object access sites opens up an opportunity to handle multiple IC misses *at the same time*. Given that the workloads that we consider execute in a fairly deterministic way, this information can be shared across executions. Second, the majority of the handlers are context-independent and, therefore, can be reused across executions.

4 Reusable Inline Caching

The goal of this paper is to reduce IC misses by reusing IC information across program executions. In an *Initial* execution of the program, the IC is built as usual. After the execution completes, in an off-line *Extraction Phase*, our runtime extracts context-independent IC information from the `ICVector` generated by the program. Later, in a subsequent execution of the program that we call *Reuse* execution, the extracted IC information is dynamically used to avert some IC misses and the generation of some handler routines. We call this scheme *Reusable Inline Caching (RIC)*. RIC is based on the three ideas of Table 2.

Linking of object access sites. The first idea is to link object access sites. The proposal is based on our observation that a hidden class created in an object access site is often later encountered in multiple other object access sites, creating an IC miss in each of these sites. This observation was hinted at by the data in Section 3.3. We call the object access site where the new hidden class is created the *Triggering* site, and the subsequent sites that encounter the hidden class and miss on it the *Dependent* sites.

In the extraction phase, the runtime examines the `ICVector` of the program, and identifies individual Triggering sites and their corresponding Dependent ones. In the Reuse execution, as soon as an IC miss occurs and a new hidden class is created, the runtime checks whether this is a Triggering site and, if so, if this site has Dependent ones. If so, after it has created an `ICVector` slot for the Triggering site, it also creates `ICVector` slots for the Dependent ones. Each Dependent's slot receives the hidden class and a handler. Note that the IC miss in the Triggering site is not averted, but when execution reaches a Dependent site, the IC miss there is averted.

Reusing handlers. The second insight is that many handlers are context-independent and, therefore, can be reused across executions. This insight is backed up by data in Section 3.3. Consequently, in the extraction phase, as the runtime examines the `ICVector`, it identifies and saves context-independent handlers. As indicated above, during the Reuse execution, when an `ICVector` slot for a Dependent site is filled, it receives a pointer to the saved handler. This averts a future IC miss. If the handler for a would-be Dependent site is not context-independent, the site is not added to the Dependent list in the extraction phase. Note that RIC does not attempt to save hidden classes, since they are context-dependent.

Hidden class validation. The third idea is to incrementally and dynamically certify that hidden classes in the Reuse run match those in the Initial run. This process is called *validation*, and ensures that reusing IC information maintains correct execution. Built-in objects are immediately marked as validated at the startup of the JavaScript runtime, since their creation is deterministic in every execution. Further, if we reach a Triggering site and the incoming hidden class is marked as validated, then RIC validates the outgoing hidden class, as the new hidden class also matches in both runs. In addition to marking this hidden class as validated, this is when RIC preloads `ICVector` slots for the Dependent sites. If a hidden class created in a Triggering site cannot be validated, then its Dependent sites cannot preload an `ICVector` slot and will therefore suffer an IC miss. Conceptually, validation confirms that the Reuse run does not diverge from the Initial run at this point. Validation often succeeds if the execution is largely deterministic, like in library initialization.

5 Implementation

RIC is implemented in two steps. First, during the extraction phase after the Initial execution, the RIC runtime analyzes the `ICVector` generated by the completed program, and creates a new structure that we call `ICRecord`. Then, during the Reuse execution, the RIC runtime uses the `ICRecord` to eagerly preload the `ICVector` of the program dynamically, and save IC misses. In the following subsections, we first describe the layout of `ICRecord`'s components. Next, we describe the RIC operations during the extraction phase and the Reuse execution. Finally, we provide an example walk-through to demonstrate RIC's operation.

5.1 ICRecord Layout

As shown in Figure 6, the `ICRecord` is a simple data structure with three components: the *Hidden Class Validation Table (HCVT)*, the *Triggering Object Access Site Table (TOAST)*, and the set of context-independent handlers generated in the Initial run.

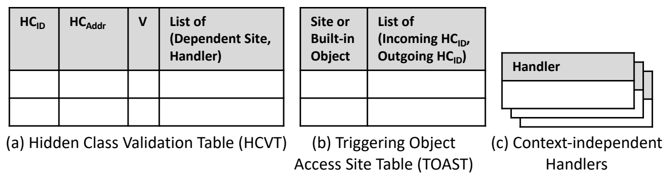

HC_{ID}
HC_{Addr}
V
List of (Dependent Site, Handler)
Site or Built-in Object
List of (Incoming HC_{ID}, Outgoing HC_{ID})
Handler
(a) Hidden Class Validation Table (HCVT)
(b) Triggering Object Access Site Table (TOAST)
(c) Context-independent Handlers

Figure 6. `ICRecord` structure.

The `HCVT` serves two purposes. First, it tracks which hidden classes are validated. Second, it provides the data needed by RIC to preload the `ICVector` slots of Dependent sites for a given Triggering site. The `HCVT` has as many entries as the number of different hidden classes created during the execution of the Initial run — including hidden classes for built-in objects such as `Object` and `Array`. For each entry, the `HCVT` contains four fields: an integer hidden class identifier (HC_{ID}), the address of the hidden class in the Reuse execution (HC_{Addr}), a *Validated (V)* bit, and a list of the Dependent sites for the Triggering site of this hidden class. For each Dependent site, the table also includes the handler to use. Recall that only sites with context-independent handlers can be considered Dependent. The `HCVT` is indexed by the HC_{ID}. HC_{ID}s are assigned by RIC during the extraction phase and act as primary keys for the `HCVT` entries.

The `TOAST` is used to initiate the process of reusing IC information in the Reuse execution. This table has as many entries as the number of Triggering object access sites in the Initial run plus the number of built-in objects in the Initial run. We include the latter because built-in objects are not created by any object access site. Recall that Triggering object access sites are object access sites that cause a hidden class transition. An entry in the `TOAST` table has two fields: one that contains the object access site ID or built-in object's name, and a second one with a list of (incoming HC_{ID}, outgoing HC_{ID}) pairs. `TOAST` is indexed by a hash generated by either the object access site ID (determined by file name, line number and position in the line) or the built-in object name string. Both are invariant across executions. In `TOAST` entries, monomorphic sites only have one incoming and one outgoing hidden class pair; polymorphic ones have multiple pairs. Entries for built-in objects have no incoming hidden class and only one outgoing hidden class.

Finally, the context-independent handlers are stored separately within the `ICRecord`. The addresses of these handlers will be stored in the `HCVT` as the `HCVT` is filled during the Reuse execution.

5.2 RIC Operation

5.2.1 Extraction Phase

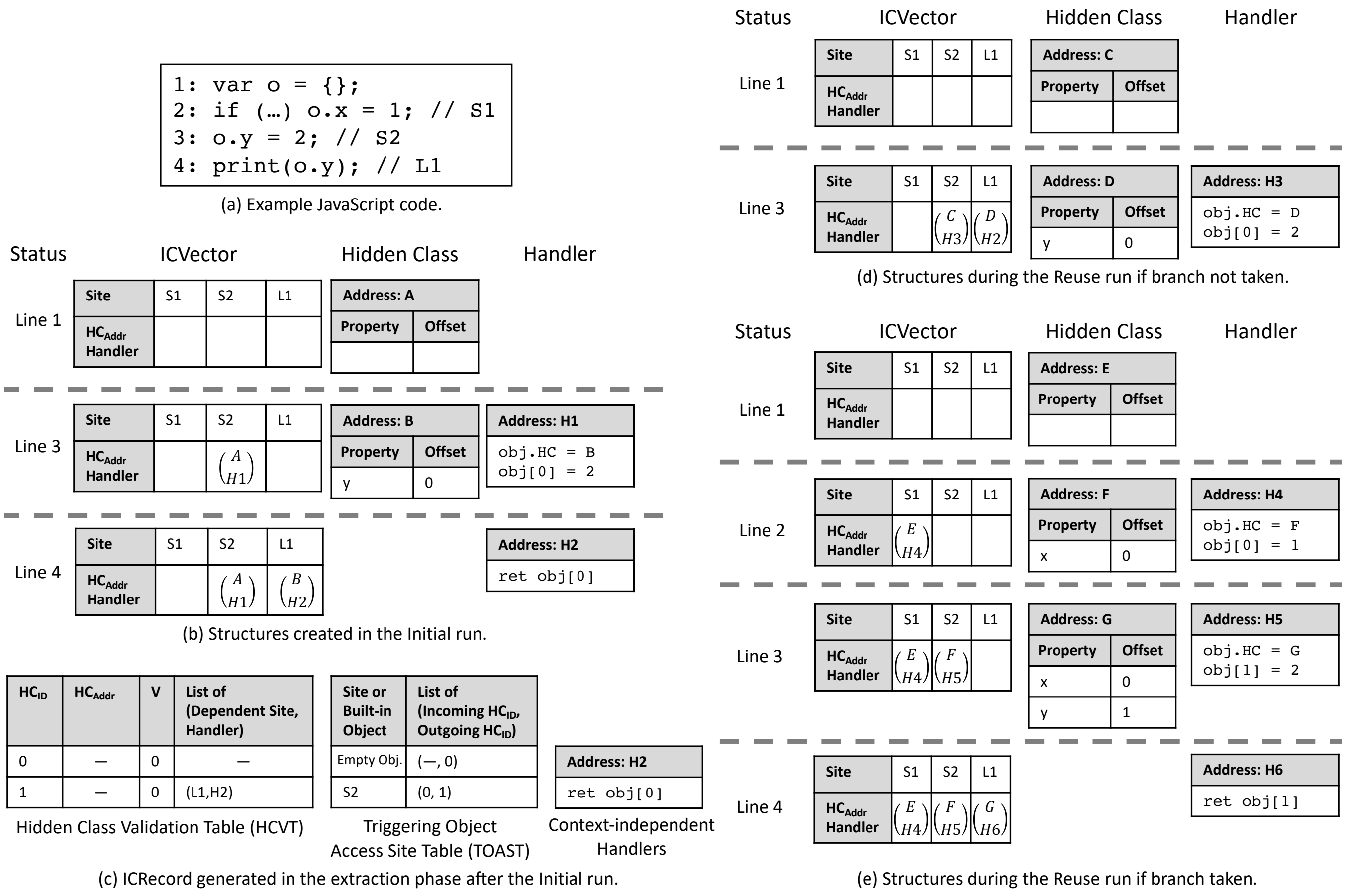
During the extraction phase, RIC creates the `ICRecord` and populates its data structures by processing the `ICVector` and hidden classes created in the Initial execution. First, RIC creates the `HCVT` and adds an `HCVT` entry for each hidden class created in the Initial execution. Monotonically increasing integers are assigned as HC_{ID}s to `HCVT` entries. The HC_{Addr} field is left empty and the Validated bit is 0, as they are to be set in the Reuse execution. The Dependent site information is filled when the `TOAST` is created, as explained next.

Second, RIC scans the `ICVector` to find all Triggering sites and their corresponding Dependent sites. Then, it creates the `TOAST` and adds a `TOAST` entry for each Triggering site and for each built-in object observed in the Initial run. In each `TOAST` entry, RIC also adds one or more (incoming HC_{ID}, outgoing HC_{ID}) pairs. Then, for each `TOAST` entry, RIC takes each outgoing HC_{ID}, locates the `HCVT` entry for the outgoing HC_{ID}, and adds the list of (Dependent site, handler) tuples to it. Lastly, RIC also saves all context-independent handlers, which are referred to by the `HCVT`.

5.2.2 Reuse Execution

In the Reuse execution, every time that a built-in object is created at the startup of the JavaScript runtime, or every time that execution encounters a Triggering object access site, the runtime checks the `TOAST` and finds the corresponding entry. If this is a built-in object, the outgoing HC_{ID} is read and used to index the `HCVT`. In the corresponding `HCVT` entry, RIC saves the address of the hidden class (HC_{Addr}) and sets the V bit. Further, it reads the entry's list of (Dependent site, handler) tuples. Next, for each tuple, it creates a new slot in the `ICVector` structure of the Reuse execution for the Dependent site. The slot is set to the built-in object hidden class and the handler. By filling one or more slots this way, we will avoid one IC miss in each of the Dependent sites.

If, instead, the access to the `TOAST` is for a Triggering object access site, RIC reads the list of (incoming HC_{ID}, outgoing HC_{ID}) pairs. At most, only one of these pairs is relevant, namely the one whose incoming HC_{ID} matches the current incoming object's hidden class at this site. Hence, for each incoming HC_{ID}, RIC accesses the `HCVT` with HC_{ID}, reads the entry's HC_{Addr} and compares it to the current incoming object's hidden class. If there is a match, then we are certain that the pair's outgoing HC_{ID} is the hidden class that will be generated at this Triggering site. Hence, RIC accesses the `HCVT` with the outgoing HC_{ID}. In the corresponding entry, it saves the outgoing hidden class address in HC_{Addr}, sets the



```
1: var o = {};
2: if (…) o.x = 1; // S1
3: o.y = 2; // S2
4: print(o.y); // L1
```

(a) Example JavaScript code.

Status | ICVector | Hidden Class | Handler

Line 1

Site	S1	S2	L1
HC_{Addr} Handler			

Address: A	
Property	Offset

Line 3

Site	S1	S2	L1
HC_{Addr} Handler		$\binom{A}{H1}$	

Address: B	
Property	Offset
y	0

Address: H1
```
obj.HC = B
obj[0] = 2
```

Line 4

Site	S1	S2	L1
HC_{Addr} Handler		$\binom{A}{H1}$	$\binom{B}{H2}$

Address: H2
```
ret obj[0]
```

(b) Structures created in the Initial run.

HC_{ID}	HC_{Addr}	V	List of (Dependent Site, Handler)
0	—	0	—
1	—	0	(L1,H2)

Hidden Class Validation Table (HCVT)

Site or Built-in Object	List of (Incoming HC_{ID}, Outgoing HC_{ID})
Empty Obj.	(—, 0)
S2	(0, 1)

Triggering Object Access Site Table (TOAST)

Address: H2
```
ret obj[0]
```

Context-independent Handlers

(c) ICRecord generated in the extraction phase after the Initial run.

Status | ICVector | Hidden Class | Handler

Line 1

Site	S1	S2	L1
HC_{Addr} Handler			

Address: C	
Property	Offset

Line 3

Site	S1	S2	L1
HC_{Addr} Handler		$\binom{C}{H3}$	$\binom{D}{H2}$

Address: D	
Property	Offset
y	0

Address: H3
```
obj.HC = D
obj[0] = 2
```

(d) Structures during the Reuse run if branch not taken.

Status | ICVector | Hidden Class | Handler

Line 1

Site	S1	S2	L1
HC_{Addr} Handler			

Address: E	
Property	Offset

Line 2

Site	S1	S2	L1
HC_{Addr} Handler	$\binom{E}{H4}$		

Address: F	
Property	Offset
x	0

Address: H4
```
obj.HC = F
obj[0] = 1
```

Line 3

Site	S1	S2	L1
HC_{Addr} Handler	$\binom{E}{H4}$	$\binom{F}{H5}$	

Address: G	
Property	Offset
x	0
y	1

Address: H5
```
obj.HC = G
obj[1] = 2
```

Line 4

Site	S1	S2	L1
HC_{Addr} Handler	$\binom{E}{H4}$	$\binom{F}{H5}$	$\binom{G}{H6}$

Address: H6
```
ret obj[1]
```

(e) Structures during the Reuse run if branch taken.

Figure 7. Example of how RIC extracts and utilizes the context-independent portion of the IC information.

V bit, and reads the list of (Dependent site, handler) tuples. Then, it uses the list of tuples to fill slots in the ICVector structure as in the case above for built-in objects. Again, the result is the elimination of IC misses.

If no match occurs for any of the incoming HC_{ID} in the TOAST entry, it means that the Reuse execution has diverged from the Initial execution. Consequently, RIC is unable to validate the outgoing hidden class of the Triggering site, and its Dependent sites will not get their ICVector slots preloaded.

5.3 Example of RIC

In this section, we build on the example of Section 2.4, and show how RIC extracts the context-independent IC information after the Initial run, and how it uses this information to avoid IC misses in the Reuse run.

5.3.1 Extraction Phase Generates ICRecord

Figures 7(a) and (b) repeat Figures 4(a) and (b) to provide the context. Figure 7(c) shows the ICRecord structure generated by RIC during the extraction phase by collecting all the context-independent information in the IC.

In the Initial execution, there are two hidden classes of interest: a hidden class for the built-in empty object and a hidden class with the *y* property. As shown in Figure 7(b), these hidden classes are allocated at addresses *A* and *B*, respectively. Let us assume that, during the extraction phase, RIC assigned them HC_{ID} 0 and 1, respectively.

Figure 7(c) shows that the HCVT contains an entry for each of these hidden classes. In each entry of the table, the address field (HC_{Addr}) is initially empty and the *V* bit is cleared. For HC_{ID} 0, the list of (Dependent site, handler) is empty. The reason is that the potential Dependent site observed during the Initial run for this Triggering site is *S2*. However, the handler for that site is *H1* (Figure 7(b)), which is not context-independent because it has address *B*. For HC_{ID} 1, the list of (Dependent site, handler) contains *(L1, H2)*. *L1* is the only Dependent site seen during the Initial run; its handler (*H2*) is context-independent. RIC stores *H2* in the ICRecord.

TOAST also contains two entries, one for the built-in empty object, and one for an object access site (*S2*) — both seen during the Initial run. The former has no incoming HC_{ID} because it is a built-in object, and its outgoing HC_{ID} is 0. The latter was a monomorphic site during the Initial run, which encountered the incoming HC_{ID} 0 and produced the outgoing HC_{ID} 1.

5.3.2 Reuse Run Is Able to Reuse State

Figure 7(d) shows the data structures in a Reuse run when the execution follows the same control flow as in the Initial run (i.e., the branch in Line 2 is not taken). In this case, RIC can use data from the `ICRecord`.

Initially, the `ICVector` is empty. When the built-in empty object is created at the startup of the JavaScript runtime, the RIC uses a hash of the built-in object name string to access the `TOAST`. It finds the first entry of the table. The outgoing HC_{ID} of this entry shows that this built-in object has HC_{ID} 0. Hence, RIC accesses the `HCVT` and finds the first entry. In the entry, it sets HC_{Addr} to *C*, which is the address of the built-in object's hidden class in the Reuse run (Figure 7(d)), sets *V* to 1, but does not find any Dependent sites. Hence, it does not update the `ICVector`.

In Line 2, the branch is not taken, as in the Initial execution. Then, on Line 3, the code suffers an IC miss in `S2`, and the runtime creates a new hidden class at address `D` and a new handler `H3` (Figure 7(d)). This IC miss cannot be avoided because handler `H3` is context-dependent. The runtime updates `S2`'s `ICVector` slot with a tuple of incoming hidden class `C` and handler `H3` (Figure 7(d)).

In addition, RIC checks the `TOAST` for `S2` and finds the (incoming HC_{ID}, outgoing HC_{ID}) pair of (0, 1). RIC then looks up the `HCVT` with HC_{ID} 0 and confirms that the corresponding entry's HC_{Addr} matches its incoming object's hidden class, and that the entry's *V* is set. Since the incoming hidden class is already validated, RIC proceeds to validate the outgoing hidden class. Specifically, RIC accesses the entry for HC_{ID} 1, stores address *D* in the entry's HC_{Addr}, sets the entry's *V*, and reads the (Dependent site, handler) tuples. In this case, there is a single dependent site, namely *L1*, with context-independent handler *H2*. Hence, RIC updates *L1*'s `ICVector` entry with a tuple of hidden class *D* and handler *H2* (Figure 7(d)). As a result, when the Reuse execution finally reaches *L1* in Line 4, an IC miss will be avoided.

5.3.3 Reuse Run Is Unable to Reuse State

Figure 7(e) shows the data structures in a Reuse run when the execution follows a different control flow than in the Initial run (i.e., the branch in Line 2 is taken). In this case, RIC cannot use data from the `ICRecord`.

The structures start as in Figure 7(d), except that the hidden class for the built-in object is allocated at address *E*. As in the previous example, when the RIC runtime accesses the `HCVT` entry of HC_{ID} 0, it sets the entry's HC_{Addr} to *E* and *V* to 1.

In Line 2, the branch is taken, and object access site *S1* is reached. The code suffers an IC miss, and the runtime creates a new hidden class at address `F` with property `x` and a new context-dependent handler `H4` (Figure 7(e)). The runtime updates `S1`'s `ICVector` slot with a tuple of incoming hidden class `E` and handler `H4`. As RIC accesses the `TOAST` with a hash of the object access site ID, it does not find any matching entry.

In Line 3, the code suffers an IC miss at object access site *S2*, and the runtime creates a new hidden class at address `G` with properties `x` and `y`, and a new context-dependent handler `H5` (Figure 7(e)). RIC accesses the `TOAST` with a hash of the object access site ID, and finds the entry for `S2`. The entry only has one (incoming HC_{ID}, outgoing HC_{ID}) pair, which is (0, 1). As RIC accesses the `HCVT` entry of HC_{ID} 0, it finds that the address in HC_{Addr} is `E`. This is different from the address of the incoming object's hidden class in `S2`, which is `F`. As a result, RIC cannot validate the outgoing hidden class *G*, and the `ICVector` is not updated.

Finally, on Line 4 of Figure 7(e), object access site `L1` is reached and an IC miss occurs. RIC is unable to avoid this IC miss. RIC fills the `ICVector` entry with *G* and *H6*. No information from the Initial run has been used in this site.

The example in this section has not used polymorphic object access sites for simplicity. Such sites are equally well-supported by RIC.

6 Experimental Setup

We evaluate RIC using Google's V8 JavaScript engine version 6.8 [8]. We gather execution times for different parts of the Initial and Reuse runs using the high precision timer available in the V8 runtime. We also instrument the V8 runtime to collect statistics on different aspects of IC operation. Furthermore, we perform some runs where we instrument the V8 runtime with Pin [22] to count the number of instructions.

In the Initial run of an application, the V8 compiler generates bytecodes from the source code and stores them in the code cache. The Reuse run uses the bytecodes from the code cache. We measure two Reuse runs: one with the original V8 runtime (*Conventional*) and one with RIC.

To evaluate RIC, we select 7 popular JavaScript libraries from various domains and measure their initialization performance. The libraries are shown in Table 3. They are written assuming that they run inside a browser, and need a global window object. Consequently, we insert a fake window object in the original source code to mimic a browser environment, and run the benchmarks in the standalone V8 shell. The Octane benchmark suite [13] handles the library execution in a standalone JavaScript shell in a similar way.

Lastly, to mimic real websites and evaluate the robustness of RIC, we create two synthetic websites that load the seven libraries in different orders. We use the first one for the Initial run and the second one for the Reuse run. This setup emulates the common scenario where RIC information is generated and utilized in different websites. In our experiments, we disable RIC for global objects, since IC information for global objects varies depending on the order in which

Table 3. Popular JavaScript libraries measured.

Library	What It Does
AngularJS [4] React [27]	Web application frameworks to support the development of single-page applications
CamanJS [9]	Library for image manipulation
Handlebars [15] Underscore [29]	Client-side template engines
jQuery [17]	Library for DOM manipulation
JSFeat [18]	Library for computer vision

the libraries are loaded. Enabling RIC for global objects adds only negligible performance overhead.

7 Evaluation

To evaluate RIC, we characterize the use of the IC, analyze RIC's performance improvements, and examine RIC's overheads.

7.1 Characterizing IC Usage

Table 1 from Section 3 hinted at the potential of RIC. It showed that, on average, a hidden class is encountered in about 5 object access sites. Hence, in the best case, RIC could avoid the misses in 4 out of these sites, since RIC does not target the first of such accesses. Table 1 also showed that, on average, about 60% of the handlers generated in the libraries are context-independent. Hence, RIC could reuse them in the Reuse run.

Table 4 shows the actual impact of RIC on the IC accesses in the Reuse run in each of the libraries. Columns 2 and 3 show the IC miss rates in the Initial and Reuse runs. We see that, thanks to the reuse enabled by RIC, the average IC miss rate drops from 49.19% in the Initial run to 24.08% in the Reuse run. This is a substantial reduction.

Table 4. IC miss rate in the Initial and Reuse runs.

Library	Initial Run	Reuse Run			
	IC Miss Rate (%)	IC Miss Rate (%)	Contribution To Miss Rate (%)		
			Handler	Global	Other
AngularJS	68.94	32.79	8.63	2.85	21.31
CamanJS	87.64	43.94	1.14	3.43	39.36
Handlebars	57.92	20.34	4.82	1.07	14.45
jQuery	48.50	29.28	6.49	1.13	21.66
JSFeat	18.96	8.16	0.18	1.82	6.16
React	18.67	3.83	1.90	0.31	1.62
Underscore	43.70	30.22	1.48	1.78	26.96
Average	49.19	24.08	3.52	1.77	18.79

The next three columns break down the IC miss rate in the Reuse run into three components. The sum of these three components is equal to the IC miss rate. The first one (*Handler*) is misses due to context-dependent handlers. Recall that RIC cannot reuse such handlers. We see that this has a relatively small effect. The second component (*Global*) is misses due to incoming hidden classes corresponding to global objects. Recall from Section 6 that RIC is disabled for global objects because they are context-dependent. This effect is very small. The last component (*Other*) is misses due to other effects. Many of these misses occur in Triggering sites, which RIC does not address by construction. This component is the dominant one.

7.2 Initialization Performance Improvements

Since every IC miss triggers the invocation of the V8 runtime, RIC's ability to reduce the IC miss rate results in a reduction in the number of instructions executed. Figure 8 compares the number of dynamic instructions executed in the Reuse runs of the Conventional and the RIC configurations. For each library, the bars are normalized to Conventional. The figure shows that, on average, RIC saves 15% of the instructions. Across libraries, the instruction count reduction is roughly correlated with the reduction in IC miss rate attained by RIC. The latter can be computed by subtracting Column 2 in Table 4 from Column 1. For example, jQuery, JSFeat, and Underscore have modest instruction count reductions because the reductions in IC miss rate attained by RIC are relatively small.

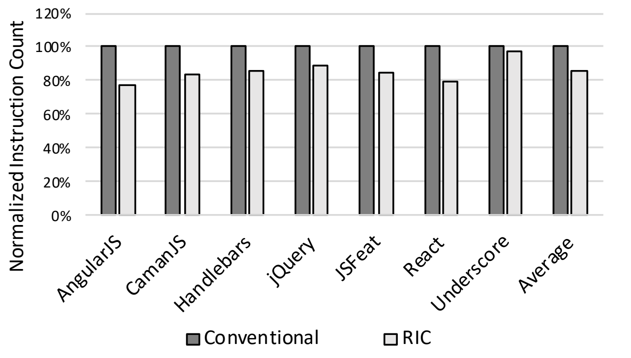


Figure 8. Instruction count in the Reuse runs for Conventional and RIC. The bars are normalized to Conventional.

This reduction in instructions executed results in a reduction in the execution time of the libraries. Figure 9 compares the execution time of the Reuse runs in the Conventional and the RIC configurations. For each library, the bars are normalized to Conventional. On top of each Conventional bar, we place the absolute time taken by the Conventional Reuse run in milliseconds.

The figure shows that the Conventional Reuse runs take between 21 and 216 ms to execute. On average, RIC reduces the execution time per workload by 17%. The largest absolute reduction is 56 ms for React. This reduction in initialization

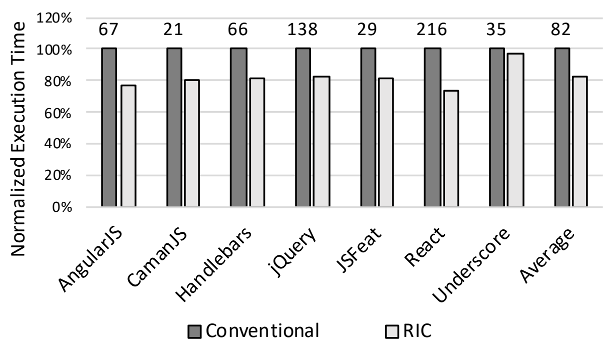


Figure 9. Execution time of the Reuse runs normalized to Conventional. On top of the Conventional bars, we show the absolute time taken by the Conventional Reuse run in milliseconds.

time is significant, as it has an impact on the experience of the user. Furthermore, as indicated in Section 1, the average number of JavaScript requests per website is increasing, and several of such requests are likely to be for libraries. In the hypothetical case that all of our seven libraries were called, RIC would reduce the initialization time by 117 ms.

Across libraries, the reduction in execution time roughly follows the reduction in instruction count in Figure 8. The time reduction is slightly higher than the instruction count reduction because the instructions eliminated involve cache misses. Hence, they are, on average, slightly more costly than the average instruction. Overall, RIC speeds-up the startup time of JavaScript codes.

7.3 Overheads of RIC

While RIC does not affect the Initial execution at all, it introduces additional computation in the extraction phase and the memory overhead of `ICRecord` in the Reuse execution. The execution time overhead of RIC's additional operations in the Reuse execution is negligible.

For our workloads, the extraction phase takes between 6 and 30 ms, with an average of 13 ms. However, it is not on the critical path, as it is executed in the background after the Initial run. Also, the current implementation of the extraction phase is naive and can be optimized.

RIC's memory overhead due to `ICRecord` in the Reuse execution ranges from 11 to 118 KB, with an average of 39 KB. For comparison, the heap usage of our libraries in the Conventional configuration is between 2.6 and 5.6 MB, with an average of 3.7 MB. Thus, RIC's memory overhead is minimal, as it accounts for only about 1% on average.

8 Discussion

In this section, we place RIC in the context of other related efforts to speed-up scripting language programs.

8.1 Reducing JavaScript Startup Overhead

Since there is no standard bytecode for JavaScript and every JavaScript compiler generates code differently, a JavaScript program is delivered in source code format. This means that, before execution can start, the program must be parsed and compiled. This can take about 30% of the total execution time in real-world websites [30].

To reduce this overhead, after a program has executed, current browsers cache its bytecode for subsequent executions. This is possible because, within a given implementation, parsing source code to its internal bytecode representation is deterministic. This technique relies on some caching APIs provided by JavaScript implementations. In particular, Google's V8 JavaScript engine provides APIs to enable host systems (e.g., web browsers) to cache the bytecode result for frequently executed JavaScript files [5]. Previously, only a limited subset of the source code's parsing result could be cached. However, recent improvements [7] have allowed the majority of the bytecode results to be reused across executions.

In the past, compilers would generate and keep the bytecode of the entire JavaScript program, in what is called *Eager* compilation. However, since only a small fraction of the program is usually executed, eager compilation is undesirable [26]. It increases memory pressure by keeping code objects for functions that are ultimately never executed. Hence, every major JavaScript compiler currently implements *Lazy* compilation. The idea is to generate and keep bytecode for only some parts of the program. Specifically, current browsers keep bytecode for the parts of the program that have been executed in a previous execution. This is reasonable since, typically, only a fraction of the functions are executed [26], and they often are the same across runs. We use this approach by default in all of our experiments.

8.2 Reusing Compilation Efforts across Executions

In programming languages other than JavaScript, there are a few advanced techniques that reuse compilation results across executions, reducing the time taken by re-executions. Most notably, HHVM [1, 25] has pioneered such efforts in PHP. HHVM is a region-based JIT compiler that uses a combination of profiling and on-stack replacement [16] to aggressively optimize a very long trace with techniques like type specialization and IC. Furthermore, it enables the reuse of the optimized traces across multiple web requests. Hence, HHVM reuses JITed code across runs.

The full potential of HHVM is achieved when running programs written in Hack, which is a dialect of PHP. One of the features not supported in Hack is the ability to add or remove properties to an object dynamically. Such restriction makes it possible to know all object type information ahead of execution. This in turn simplifies code generation for object access sites, and enhances the reuse of IC information across executions. Still, Hack does support dynamic method dispatch, and HHVM optimizes it with IC and reuses this information across executions.

When running programs written in the original PHP language, HHVM does support dynamic properties. However, it relies on expensive hash table lookups every time a dynamic property is read or written [32]. In JavaScript, every property is a dynamic property, and it would be prohibitively expensive to use a hash table lookup like HHVM for every object access.

For Java, in the context of the Android runtime, ShareJIT by Xu et al. [31] proposes a global code cache for JIT compilers that can share optimized code across multiple applications and multiple processes. ShareJIT constrains the level of context-specific optimization (e.g., the scope of inlining) to make it feasible to share optimized code between executions. Also, in the context of data-parallel systems based on the Java Virtual Machine (JVM), Lion et al. [21] propose a technique to reuse a pool of JVMs across multiple applications to avoid startup overhead such as class loading and bytecode interpretation. Java does have dynamic method dispatch, and a VM like ShareJIT optimizes it with IC and reuses IC information across executions. However, similar to Hack, Java does not support dynamic properties. RIC mainly focuses on the overhead stemming from supporting dynamic properties.

9 Related Work

Ahn et al. [2] propose a modified type system that decouples the prototype pointer from hidden classes, and enables the hidden class and IC information to be reused when refreshing the same website. Its applicability, however, is somewhat limited, as it lacks a mechanism to make the information persistent, and improves the performance only under specific conditions (e.g., reloading the same website before garbage collection and without restarting the browser). In addition, changing the hidden class structure may hurt other optimization techniques, which require the stronger type guarantee provided by the prototype pointer. Our proposal is more generally applicable and does not interfere with other optimization techniques, as it does not change the hidden class structure.

Oh and Moon [24] propose a snapshot technique to accelerate JavaScript initialization for mobile applications. V8 also has a similar API to create a custom startup snapshot [6]. Such approaches take a snapshot of the heap after the initialization of JavaScript frameworks and libraries. When the same application is invoked later, the runtime restores the objects from the snapshot to the heap, instead of recreating them by executing JavaScript code for initialization. While the snapshot technique may greatly reduce the startup time by completely avoiding JavaScript initialization, it has several limitations. First, it is too rigid for different applications to share information. Given two applications that use the same JavaScript library, a snapshot is application-specific, and each application has to create its own snapshot. In contrast, in RIC, the information is maintained for each JavaScript file. Therefore, the IC information for a library can be shared by different applications. Second, the snapshot technique can be incorrect if the initialization involves any non-deterministic behavior, such as the use of the date function or I/O operations. Again, in RIC, the initialization code is still fully executed, but is accelerated with the hints from the previous execution. It produces correct results even if the initialization has non-deterministic behavior.

There are some server-side techniques to reduce page load time [14, 20]. Kedlaya et al. [20] propose server-side type profiling to optimize client-side JavaScript engines. They use server-side profiling to identify functions with dynamic behavior that are frequently deoptimized and waste computing power. Then, they mark such functions so that the client-side JavaScript engine does not attempt to optimize them. Compared to our proposal, their technique conveys its information as source code comments, and is orthogonal to our approach. Google's PageSpeed [14] is an open-source Apache HTTP Server module which automatically applies best practices to website source code, including CSS, JavaScript, and images. It has more than 40 available optimization techniques, including CSS and JavaScript concatenation, inlining, image optimization, and resizing. PageSpeed can be used to complement our approach to reduce page load time.

10 Conclusion

In this paper, we reduced the startup time of JavaScript programs by enhancing the reuse of compilation and optimization information across different executions. We call our scheme *Reusable Inline Caching (RIC)*. RIC is based on two observations: (i) a hidden class created in an object access site is often later encountered in multiple other object access sites, creating an IC miss in each of these sites, and (ii) the majority of handler routines are context-independent. RIC uses these observations to extract context-independent information from the IC and reuse it in subsequent runs. As a result, RIC minimizes the number of IC misses, which reduces the startup time of the program. We integrated RIC into the state-of-the-art Google V8 JavaScript engine and measured its impact on the initialization time of popular JavaScript libraries. On average per library, RIC reduced the instruction count by 15%, and the execution time by 17%.

Acknowledgments

This work was funded by NSF under grant NSF CCF 15-27223. We thank Martin Maas for his extensive feedback.

References

[1] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming, Systems, Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 777–790. https://doi.org/10.1145/2714064.2660199

[2] Wonsun Ahn, Jiho Choi, Thomas Shull, Maria J. Garzaran, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

[3] Akamai Technologies. 2014. Performance Matters: 9 Key Consumer Insights.

[4] AngularJS. 2019. https://angularjs.org/

[5] Chromium Blog. 2015. New JavaScript techniques for rapid page loads. https://blog.chromium.org/2015/03/new-javascript-techniques-for-rapid.html

[6] V8 Blog. 2015. Custom startup snapshots. https://v8.dev/blog/custom-startup-snapshots

[7] V8 Blog. 2018. Improved code caching. https://v8.dev/blog/improved-code-caching

[8] V8 Blog. 2019. V8 JavaScript Engine. https://v8.dev/

[9] CamanJS. 2016. http://camanjs.com/

[10] Craig Chambers, David Ungar, and Elgin 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, New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

[11] L. Peter Deutsch and Allan M. 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, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[12] MDN Web Docs. 2019. Standard built-in objects. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects

[13] Google. 2017. Octane Benchmark. https://developers.google.com/octane/

[14] Google. 2018. PageSpeed Module. https://developers.google.com/speed/pagespeed/module/

[15] Handlebars. 2019. http://handlebarsjs.com/

[16] Urs Hölzle, Craig Chambers, and David Ungar. 1992. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*. ACM, New York, NY, USA, 32–43. https://doi.org/10.1145/143095.143114

[17] jQuery. 2019. http://jquery.com

[18] JSFeat. 2018. http://inspirit.github.io/jsfeat/

[19] Jupiter Research. 2006. Retail web site performance: Consumer reaction to a poor online shopping experience.

[20] Madhukar N. Kedlaya, Behnam Robatmili, and Ben Hardekopf. 2015. Server-side Type Profiling for Optimizing Client-side JavaScript Engines. In *Proceedings of the 11th Symposium on Dynamic Languages (DLS 2015)*. ACM, New York, NY, USA, 140–153. https://doi.org/10.1145/2816707.2816719

[21] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski, and Ding Yuan. 2016. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. USENIX Association, Savannah, GA, 383–400.

[22] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. ACM, New York, NY, USA, 190–200. https://doi.org/10.1145/1065010.1065034

[23] Bryan McQuade, Doantam Phan, and Mona Vajihollahi. 2013. Instant Mobile Websites: Techniques and Best Practices. Google I/O '13.

[24] JinSeok Oh and Soo-Mook Moon. 2015. Snapshot-based Loading-time Acceleration for Web Applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 179–189. https://doi.org/10.1109/CGO.2015.7054198

[25] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 151–165. https://doi.org/10.1145/3192366.3192374

[26] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proceedings of the USENIX Conference on Web Application Development (WebApps'10)*. USENIX Association, Berkeley, CA, USA, 3–3.

[27] React. 2019. https://facebook.github.io/react/

[28] Carroll Rheem. 2010. Consumer Response to Travel Site Performance. https://www.phocuswright.com/Free-Travel-Research/Consumer-Response-to-Travel-Site-Performance. PhoCusWright.

[29] Underscore. 2018. http://underscorejs.org/

[30] Toon Verwaest and Camillo Bruni. 2016. Real-world JavaScript Performance. https://docs.google.com/presentation/d/14WZkWbkvtmZDEIBYP5H1GrbC9H-W3nJSg3nvpHwfG5U/edit?usp=sharing

[31] Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. 2018. ShareJIT: JIT Code Cache Sharing Across Processes and Its Practical Implementation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '18)*. ACM, New York, NY, USA, 124:1–124:23. https://doi.org/10.1145/3276494

[32] Owen Yamauchi. 2015. *Hack and HHVM: Programming Productivity Without Breaking Things* (1st ed.). O'Reilly Media, Inc.

[33] Zona Research. 1999. The Economic Impacts of Unacceptable Web-Site Download Speeds. www.webperf.net/info/wp_downloadspeed.pdf