Improving JavaScript Performance by Deconstructing the Type System[*]

Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas
University of Illinois at Urbana-Champaign
{dahn2, jchoi42, shull1, garzaran, torrella}@illinois.edu

ABSTRACT

Increased focus on JavaScript performance has resulted in vast performance improvements for many benchmarks. However, for actual code used in websites, the attained improvements often lag far behind those for popular benchmarks.

This paper shows that the main reason behind this shortfall is how the compiler understands types. JavaScript has no concept of types, but the compiler assigns types to objects anyway for ease of code generation. We examine the way that the Chrome V8 compiler defines types, and identify two design decisions that are the main reasons for the lack of improvement: (1) the inherited prototype object is part of the current object's type definition, and (2) method bindings are also part of the type definition. These requirements make types very unpredictable, which hinders type specialization by the compiler. Hence, we modify V8 to remove these requirements, and use it to compile the JavaScript code assembled by JSBench from real websites. On average, we reduce the execution time of JSBench by 36%, and the dynamic instruction count by 49%.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Very high-level languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects, Inheritance, Polymorphism*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Compilers, Optimization*

General Terms

Design, Languages, Performance

Keywords

JavaScript, Scripting Language, Dynamic Typing, Type Specialization, Hidden Class, Inline Caching, Prototype

[*]This work is supported in part by NSF under grants CCF-1012759, CNS-1116237, and CNS-1319657, and Intel under the Illinois-Intel Parallelism Center (I2PC).

PLDI'14, June 09 - 11 2014, Edinburgh, United Kingdom.
Copyright 2014 ACM 978-1-4503-2784-8/14/06 ...$15.00.
http://dx.doi.org/10.1145/2594291.2594332

1. INTRODUCTION

The vast majority of websites today use JavaScript [2] to implement their client-side logic. This trend seems set to accelerate with the introduction of new web standards such as HTML5 that allow JavaScript to handle more of the dynamism in the web. At the same time, the advent of mobile computing and, more recently, wearable computing means that processors necessarily have to get simpler. This has put a spotlight on JavaScript performance, despite hard-to-optimize features such as dynamic typing.

As a result of this pressure, advanced JavaScript compilers such as Chrome V8 [8] have recently shown great improvement. However, most of the effort has been focused on speeding-up a set of popular benchmarks — e.g., Kraken [4], Octane [5], and SunSpider [7]. Only recently has it come to light that the behavior of JavaScript code in real websites is more dynamic than in these benchmarks [20, 23, 22]. This dynamism presents new challenges to the compiler.

In this paper, we focus on understanding and eliminating the main performance bottlenecks of JavaScript code in real websites. We start by showing that V8 is unable to substantially improve the performance of JavaScript in JSBench [21] — a suite that assembles code from some real websites. Next, we show that a key reason behind this shortfall is the higher degree of type dynamism in website code, which makes it hard for the compiler to predict object types and generate specialized code for them.

We show that this type dynamism stems from the way the compiler understands types. Types do not exist in JavaScript as a language concept, but the compiler imposes types anyway to describe objects for the purposes of type specialization. V8 defines three aspects of objects to be part of the type description: (1) the object's structure, (2) the object's inherited prototype, and (3) the object's method bindings. This seems natural, since this is how statically-typed object-oriented languages also define types. The expectation is that, although JavaScript allows for more dynamism, it will follow the behavior of static languages in actual execution. This assumption, while true for popular benchmarks, often turns out to be inaccurate for real website code. Based on this observation, we propose and evaluate three enhancements to the V8 compiler. These enhancements decouple prototypes and method bindings from the type definition. As a result, they eliminate most type unpredictability and speed-up JavaScript execution substantially.

The contributions of this paper are:
• It characterizes, for the first time, the internal behavior of the V8 compiler when executing website code (as assembled by JSBench) compared to popular benchmarks. V8 is on the cutting edge of JavaScript compilation.

• It identifies type unpredictability as the main source of performance shortfall for website code. Further, it singles out frequent changes to prototypes and method bindings as the cause of the unpredictability, as well as scenarios that trigger them.
• It proposes three enhancements to the compiler that effectively decouple prototypes and method bindings from the type definition, and eliminate most type unpredictability.
• It implements these enhancements in V8 and evaluates them with JSBench. The results show that, on average, our enhancements reduce the execution time by 36%, and the dynamic instruction count by 49%. Moreover, the reduction in type dynamism leads to a reduction in bookkeeping information in the compiler, leading to a savings of 20% in heap memory allocation. Finally, the performance of popular JavaScript benchmarks is largely unchanged.

This paper is organized as follows: Section 2 gives a background; Section 3 motivates this work; Section 4 introduces the problem we want to solve; Section 5 presents our proposed modifications to the compiler; Section 6 discusses various aspects of the enhancements; Section 7 evaluates the enhancements; and Section 8 considers related work.

2. BACKGROUND

2.1 The JavaScript Language

JavaScript [2] is a dynamically-typed language, where variable types are not declared and everything, including functions and primitives, may be treated as an object. An object is a collection of properties, where a property has a name and a value. The value of a property can be a function, which is then referred to as a method of the object.

JavaScript uses prototyping to emulate the class inheritance of statically-typed languages such as C++ or Java. Every object in JavaScript inherits from a "parent" object, which is the prototype object of the constructor function used to create that object. Inheritance is supported by having the special property `__proto__` in the object pointing to the prototype object at object construction time. The prototype object is inherited by all the objects that are created using the corresponding constructor. Properties in the prototype can either be accessed via inheritance through the "child" object, or directly through the `prototype` property in the constructor function object.

Figure 1(a) shows a JavaScript example where a constructor function `Point` is called twice to create objects `p1` and `p2`, with properties `x` and `y`. In line 8, the code shows the use of the `prototype` property of constructor `Point` to add the property `size` with value `100` to the prototype object that has been inherited by `p1` and `p2`. Notice that when trying to print the `size` property of `p1` in line 7, it appears as undefined, as this property has not been defined yet. However, after defining it in the prototype object, both `p1` and `p2` have access to that property in lines 9 and 10.

Generating efficient code for dynamically-typed languages is difficult. For instance, suppose that we are executing the code in Figure 1(b) that accesses property `x`. Since JavaScript programmers do not declare types, it is unknown where property `x` is stored in memory. Indeed, property `x` can be found at different memory offsets for the different objects that `getX` can receive as arguments. Thus, in JavaScript, each property access requires a lookup in a dictionary to resolve the property's location in memory. In languages such as C++ or Java, properties are located at fixed offsets that can be statically determined by the compiler based on the object's class, and a property access requires a single load or store.

```
1  function Point (x, y){
2     this.x = x;
3     this.y = y;
4  }
5  var p1 = new Point (11, 22);
6  var p2 = new Point (33, 44);
7  print( p1.size ); //undefined
8  Point.prototype.size = 100;
9  print( p1.size ); //100
10 print( p2.size ); //100
```
(a)

```
1  function getX(p){
2     return p.x;
3  }
4  var sumX += getX(p1);
```
(b)

Figure 1: Example JavaScript code to create an object (a) and to access an object property (b).

Modern Just-In-Time (JIT) compilers address this problem by using specialization [10, 11]. Specialization is based on the empirical evidence that, at run time, programs are not as dynamic as they could be: the type of an object tends not to change, and the type of a variable at a given access site tends to be always consistent. Thus, JIT compilers collect profile information at runtime that they use to generate specialized code, usually guarded by a runtime condition that checks that the assumption made to generate the specialized code is correct. If the check fails, the code needs to bail out to the runtime or jump to more inefficient code.

2.2 The Chrome V8 Compiler

V8 [8] is the JavaScript JIT compiler used in the Google Chrome web browser. In the following, we describe *hidden classes* and *inline caching*. More details can be found in website documents [12] and in the V8 source code [8]. These two concepts are not specific to V8, however, and are used in all popular JavaScript compilers [1, 3, 6].

2.2.1 Hidden Classes

V8 uses *hidden classes* to introduce the notion of types. The basic idea is similar to the maps in Self [11]. Objects that are created in the same way are grouped in the same hidden class. V8 uses the information collected in the hidden class to generate efficient code, as hidden classes record the offset where a given property for that object type is located. Note that the concept of types itself is an idea imposed by the compiler for the purposes of code generation, and is not present in the semantics of a dynamically-typed language like JavaScript. For this reason, types in dynamically-typed languages are called hidden classes, to stress that the types are hidden from the programmer. We use the terms type and hidden class interchangeably.

Figure 2 shows how V8 creates hidden classes for the code in Figure 1(a). The first time that constructor `Point` is instantiated to create object `p1` in line 5, an empty Point object is created (not shown in Figure 2). Such object only contains a pointer to its hidden class, which initially is also empty (② in Figure 2). The hidden class only contains the `__proto__` pointer to the prototype of the object (③). Initially, the prototype object is also empty and contains a pointer to its own hidden class (④). When property `x` is added to `p1` in line 2, object `p1` points to a new hidden class that records the new property `x` and its offset (⑤). The value for `x`, 11 in this case, is added to the object. Similarly, when property `y` is added (line 3), the object points to a new hidden class (⑥) that contains offsets for `x` and `y`. The resulting object `p1` is now shown as ① in Figure 2. When object `p2` in line 6 is instantiated using the same constructor, another object is created, which follows the same hidden class transitions as `p1`, and ends up as object ⑦ in Figure 2.

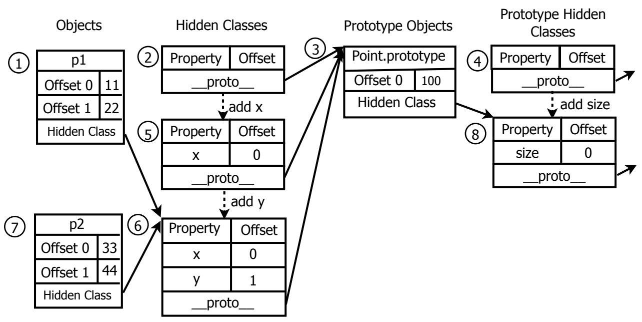

Figure 2: Example of hidden classes.

The prototype object (③) is a regular JavaScript object. Thus, when it is created, V8 also creates its hidden class (④). As usual, when a new property is added to the prototype object (`Point.prototype.size` in line 8), the prototype object transitions to a new hidden class (⑧). The prototype hidden class has a pointer to its own prototype object, which is the built-in object `Object.prototype` (not shown in the figure). Figure 2 shows the state of the heap after all the instructions in Figure 1(a) have executed.

Hidden classes are *immutable* data structures. Thus, if an object adds a new property or changes its prototype object, its `Hidden Class` pointer is updated to point to a different hidden class. Thanks to this immutability, checking if two objects have the same type is as simple as checking if the `Hidden Class` in both objects points to the same address.

2.2.2 Inline Caching

A common assumption is that property accesses at a given access site are usually performed on objects with the same type. Hence, V8 uses a technique called inline caching [13] to optimize accesses to object properties at an access site.

There are three main types of inline caches: load inline cache, store inline cache, and call inline cache. A *load inline cache* is used on a load of an object property. Figure 3 shows an example of a load inline cache for the access to property `p.x` of Figure 1(b). The load inline cache checks if `p` is an object. If it is, then it checks if the hidden class of object `p` is the same as the cached hidden class, which is the hidden class seen earlier at this site. If so, there is an inline cache hit, and the property can be accessed with a simple load using the cached offset. However, if the code encounters a type it has not seen before, it calls the runtime, which will add another entry in the code (e.g., with a *case* statement) to handle the property access for the new type. This results in inefficient code, as a property access is now forced to perform a lookup for the correct entry in the inline cache. An access site that has only seen a single type is called *monomorphic*; if it has seen multiple types, it is *polymorphic*.

A *store inline cache* is used on an update of an object property, as in lines 2 and 3 of Figure 1(a). In case of an inline cache hit, the property is updated with a simple store. Otherwise, there is an inline cache miss and the runtime is invoked. If a new property is added to the object a new hidden class must also be created. Finally, a *call inline cache* is used when an object method is called. Call inline caches are similar to load inline caches in that they need to first load the method value, which is a function object, before calling it.

```
function getX(p){
  if(p is an object &&
       p.hiddenclass==cached_hiddenclass)
    return p[cached_x_offset];
  else {
    // jump to V8 runtime
  }
}
```

Figure 3: Inline cache example for the code in Figure 1(b).

Whether an inline cache access hits or misses has performance implications. Our experimental results show that a monomorphic inline cache hit requires about 10 instructions (or only 3 instructions if optimized by the V8 Crankshaft optimizing compiler), while a polymorphic inline cache hit requires approximately 35 instructions if there are 10 types, and approximately 60 instructions if there are 20 types. An inline cache miss requires 1,000–4,000 instructions.

3. MOTIVATION OF THE PAPER

Figure 4 compares the number of instructions executed by several JavaScript benchmarks with different V8 optimization levels. We show data for the JSBench [21], Kraken [4], Octane [5], and SunSpider [7] suites. JSBench is a suite that assembles JavaScript code from some real websites, while Kraken, Octane, and SunSpider are popular benchmarks that were developed by the web browser community to compare the performance of JavaScript compilers. From left to right, we show bars for the nine individual websites in JSBench, and then the arithmetic mean for JSBench, Kraken, Octane, and SunSpider. Note that the Y axis is in *logarithmic* scale. For each benchmark (or average), we show three bars, corresponding to three environments. *Baseline* is when all the V8 optimizations have been applied. *No Crankshaft* is Baseline with the V8 Crankshaft compiler disabled. Crankshaft is the V8 optimizing compiler, which performs optimizations such as inlining, global value numbering, and loop invariant code motion, among others. Finally, *No Crankshaft, No IC* is *No Crankshaft* with inline caching disabled. For a given benchmark, the bars are normalized to *Baseline*. For all benchmarks, sufficient warm-ups were done to enable all compiler optimizations before measurement.

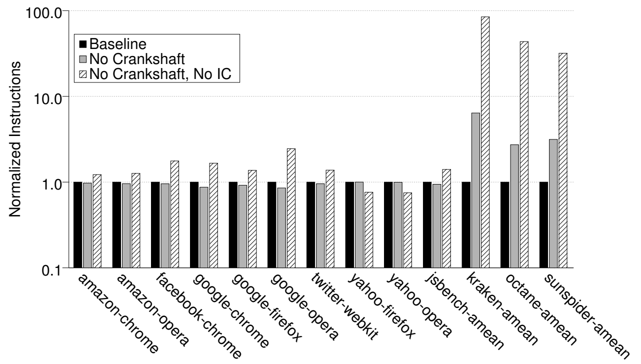

Figure 4: Comparing the number of instructions executed with different V8 optimization levels.

Looking at the mean bars, we see that both the optimizing compiler and inline caching substantially benefit the Kraken, Octane, and SunSpider suites. However they have a small or negligible effect on the JSBench suite. Specifically, disabling the optimizing compiler increases the average instruction count by 6.4x, 2.7x, and 3.1x in Kraken, Octane, and SunSpider, respectively, while it decreases the count by 5% in JSBench. Moreover, disabling the optimizing compiler and inline caching increases the average instruction count by 84.6x, 43.6x, and 31.9x in Kraken, Octane, and SunSpider, respectively, and by only 1.4x in JSBench. All the benchmarks in a given suite have a relatively similar behavior; we do not show individual benchmarks for three suites due to space limitations. In the rest of the paper, we explain why real websites as assembled in JSBench do not benefit from the V8 optimizations, and propose how to solve the problem.

4. LOW TYPE PREDICTABILITY

Websites perform poorly compared to popular benchmarks because of *type unpredictability.* We define type predictability in terms of (1) the ability of the compiler to anticipate the type of an object at the object access site, and (2) the variability in object types observed at the access site. We refer to the former form of predictability as *type-hit-rate* and the latter as *polymorphism.* Type predictability is crucial to generating high-quality code.

With a low type-hit-rate, the compiler is frequently forced to perform an expensive dictionary lookup on an object property access, instead of a simple indexed access. Also, when a new property needs to be added to the object (e.g., `x` to `p1` and then `y` to `p1` in Figure 2), the result is a type miss that involves the creation of a new hidden class — which is even more expensive. Finally, with high polymorphism, the compiler is forced to do a lookup for the entry of the correct type among multiple code entries.

There is an abundance of literature to help JavaScript programmers write high-performance code by avoiding type unpredictability. Programmers are advised to coerce all the objects at an access site to have the same set of properties, and to add the properties in the same order, such that they do not end up having different hidden classes. This is because assigning new hidden classes to objects instead of reusing old hidden classes leads to type unpredictability. First, it decreases the type-hit-rate for all the inline caches visited by the objects, since the initial sightings of the new hidden class in the inline caches result in a miss. Second, it increases the polymorphism in the same inline caches, since the new hidden class has to be added to the list of anticipated types.

However, we have discovered that, in reality, the bulk of type unpredictability in JavaScript code in websites comes from two unexpected sources: *prototypes* and *method bindings.* As mentioned before, prototypes in JavaScript serve a similar purpose as class inheritance in statically-typed object-oriented languages such as C++ and Java. Object methods serve a similar purpose as class methods in statically-typed languages. In a statically-typed language, the parent classes and the class method bindings of an object are set at object creation time and *never* change. JavaScript compilers optimize code under the assumption that, although JavaScript is a dynamically-typed language, its behavior closely resembles that of a statically-typed language.

For the popular JavaScript benchmarks that compiler developers compete on, the assumption that the prototypes and method bindings of an object almost never change holds true. However, the behavior of the code that is being used in websites is much more dynamic. In websites, prototypes and method bindings do change quite often. This results in an increase in the number of hidden classes and, along with it, type unpredictability. This is the prime reason behind the comparatively lackluster performance of the optimizing compiler and inline caching in websites seen in Figure 4.

In the following sections we describe specifically what patterns result in this behavior.

4.1 First Culprit: Prototypes

The first culprit behind the type unpredictability seen in website JavaScript code is integrating the prototype of an object into its hidden class. The immutability of the prototype in the hidden class allows the prototype to be checked automatically when the hidden class is checked in inline caches. However, the downside is that, to maintain the immutability of the prototype, a *new hidden class* needs to be created for every change in the prototype.

This is analogous to the need for the compiler to create a new hidden class each time a new property is added. There is a key difference, however. In a given piece of code, there is a fundamental limit to how many new hidden classes can be created due to property addition. This is because objects can only have so many properties. After several executions, all properties that can be added would have been added already, and the number of hidden classes would saturate very soon.[1]

However, there is *no limit* to the number of hidden classes that can be created due to changes in prototypes. In fact, new hidden classes can still be created even after multiple warm-ups of the same code. Next, we describe two cases where changes in prototypes can lead to rampant hidden class creation and type unpredictability.

4.1.1 Prototype Changes due to Function Creation

JavaScript semantics dictate that every time a function is created, a corresponding prototype for that function is created, when the function is used as a constructor. This is how JavaScript implements inheritance. The problem occurs when there is frequent function creation. Since the `__proto__` pointer is immutable in a hidden class, growth in the number of prototypes leads to a corresponding growth in the number of hidden classes.

Figure 5(a) shows an example of a loop that creates a new function object at each iteration, and assigns it to the variable `Foo`. Also at each iteration, a new object `Obj` is constructed using that function. The store to `this.sum` at line 3 is handled using an inline cache. This type of code pattern, where functions are created dynamically in the same scope as the call site, is quite common in JavaScript code — often simply because it is *easier to program* that way. Also, it leads to better encapsulation, compared to defining the function in the global scope and polluting the global name space. Regardless of the reason, ideally we would like only a single type to reach the inline cache, namely the initial hidden class created by function `Foo`.

In reality, the type of `Obj` that reaches line 3 turns out to be different at each iteration, due to the dynamic function creation. Figure 6(a) shows the state of the heap at the end of each iteration. At iteration 1, immediately af-

[1]Properties added through true dictionary accesses, in the form of `object[key]`, are an exception to this rule. However, true dictionaries are not candidates for type specialization and do not need hidden classes.

```
1 for(var i = 0; i < 100; i++) {
2   var Foo = function(x, y) {
3     this.sum = x + y;
4   }
5   var Obj = new Foo(1, 2);
6 }
```

(a)

```
1 var initAmznJQ = function() {
2   window.amznJQ = new function() {
3     var me = this;
4     me.addLogical = function(...) {...};
5     me.declareAvailable = function(...) {...};
6     ...
7   }();
8   window.amznJQ.declareAvailable(”JQuery”);
9 }
```

(b)

Figure 5: Examples where dynamic function creation leads to inline cache misses: a contrived code for illustration purposes (a), and real code in the Amazon.com website (b).

ter `Foo` is called as a constructor in line 5, V8 creates an object (① in Figure 6(a) but initially empty), along with a hidden class (②) for the object with `__proto__` pointing to `Foo.prototype` (③). When the inline cache in line 3 is reached, it sees hidden class ② for the first time, and so it misses. The runtime miss handler creates a new hidden class (④) with the new property `sum`. Also, the value of `sum` is added at the correct offset in object ①, and the object's type is updated to be the new hidden class ④.

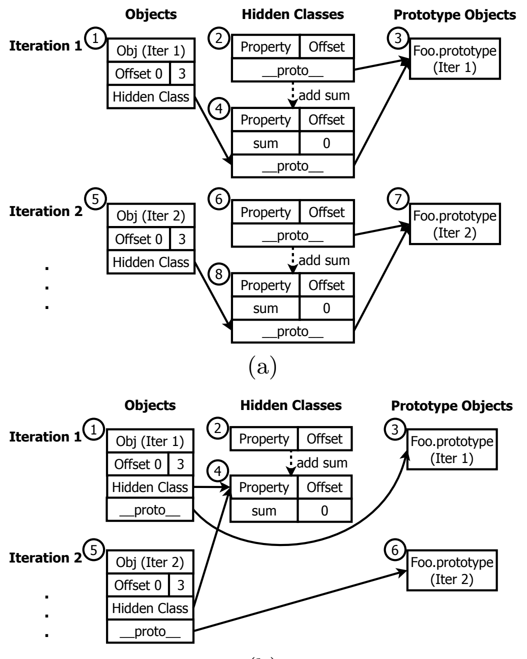

Figure 6: State of the heap after executing iterations of the loop in Figure 5(a): using the current V8 design (a), and after restructuring V8 (b).

At iteration 2, when constructor `Foo` is called, another empty object is created (⑤ but empty). Unfortunately, it *cannot reuse* the old hidden class ②, and a new hidden class (⑥) has to be created. The reason is that a new prototype object (⑦ in this case) has to be created when a new function is created, and the `__proto__` pointer in the hidden class ② is immutable. Therefore, when the inline cache in line 3 is reached, it has never seen hidden class ⑥ before. The inline cache misses again and object ⑤ transitions to the new hidden class ⑧. Now, the inline cache turns polymorphic. Overall, a new hidden class is assigned to `Obj` at every iteration, and the inline cache in line 3 misses every time.

Figure 5(b) shows a similar example, but this time from actual JavaScript code in the Amazon.com website taken from the JSBench benchmarks [21]. At line 2, object `window.amznJQ` is constructed using the anonymous function defined between lines 2-7. The actual call to the function is at line 7, at the end of the definition. A new instance of the function is created at every call to `initAmznJQ` and hence, object `window.amznJQ` ends up having a different hidden class at every invocation due to different prototypes. As a result, the store inline caches for `me.addLogical` (line 4) and `me.declareAvailable` (line 5) inside the constructor miss every time, as well as the call inline cache for `window.amznJQ.declareAvailable` (line 8).

4.1.2 *Prototype Changes due to Built-in Object Creation*

JavaScript specifies a set of built-in functions with which to construct objects with pre-defined support for some primitive operations. These built-in functions include `Object(...)`, `Function(...)`, `Array(...)`, and `String(...)`, among others. The primitive operations are implemented as properties in the prototypes of these functions, and objects constructed using these functions gain automatic access to them through inheritance. Figure 7 shows the inheritance graph for these built-in prototypes and their relationship with user-created objects. The dotted arrows denote the inheritance relationships. All prototype inheritance chains eventually converge to `Object.prototype`, which has the `null` value as its prototype.

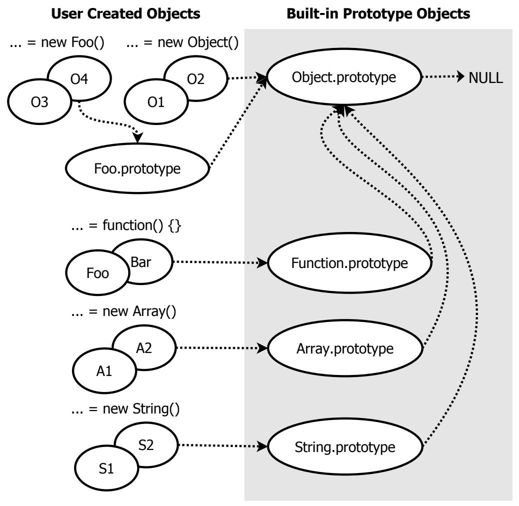

Figure 7: Inheritance graph of built-in objects.

The second source of rampant hidden class creation and type unpredictability is the regeneration of these built-ins from scratch. One might think that updates to built-in objects would be rare. However, there is one frequent web browser event that triggers such updates: *Page Loads.* Whenever a page is loaded, semantically JavaScript must create a new global execution context. The context contains new instances of the built-ins described above because JavaScript permits modification of even built-in objects in the course of user-code execution. This means that even when an identical page is *reloaded* in the course of a web browser session, it must re-create all the built-in objects.

Regeneration of the built-in prototype objects causes all objects that were constructed using the built-in functions to have different hidden classes, due to the immutability of the `__proto__` property in hidden classes. It also causes automatically-generated function prototypes that inherit from `Object.prototype` to have different hidden classes. This leads to type unpredictability and, hence, low performance.

4.2 Second Culprit: Method Bindings

The second culprit behind type unpredictability is encoding the bindings of the methods of an object into its hidden class. The method bindings of an object never change in a statically-typed language. However, functions are first-class citizens in JavaScript, and the value of a function property can change at any time during the lifetime of an object. In spite of this, the V8 compiler assumes that method bindings stay mostly static in actual code and optimizes for this case. However, in real JavaScript code in websites, this assumption does not hold, and causes added type unpredictability.

Function bindings are encoded into hidden classes in V8 by storing the values of function properties in the hidden class itself, rather than the offsets in the object like regular properties. These values are immutable, just like offsets are immutable in a hidden class. This immutability allows method bindings to be checked automatically when hidden classes are checked in call inline caches. However, immutability also means that, if the method binding changes, a new hidden class has to be created. This gives rise to the same problem as with prototypes: frequent changes in the method binding lead to excessive hidden class creation and type unpredictability.

To hedge against the possibility of frequent method binding changes, V8 gives up on storing method bindings as soon as it encounters a change in the value of a function property. Then, V8 stores the function property in the object, just like other properties. However, even this hedging results in significant type unpredictability. To see why, consider the example in Figure 8. The figure shows a loop that creates a new object `Obj` at each iteration, and then assigns new functions to properties `Foo` and `Bar`. The call to function property `Foo` at line 5 is made using a call inline cache. We would like that only a single type reaches the inline cache.

```
1 for(var i = 0; i < 100; i++) {
2   var Obj = new Object();
3   Obj.Foo = function() {};
4   Obj.Bar = function() {};
5   Obj.Foo();
6 }
```

Figure 8: Example where assigning a function to an object property leads to type unpredictability.

In reality, the type of `Obj` that reaches line 5 only stabilizes after multiple iterations of the loop. This can seen in Figure 9(a), which shows the state of the heap at the end of each iteration of the example loop. This time the `__proto__` properties in hidden classes are omitted for brevity.

At iteration 1, in line 2, `Obj` is initially created as an empty object (①) and a new hidden class (②) is also created for that object. Initially, ① points to ②. In line 3, when

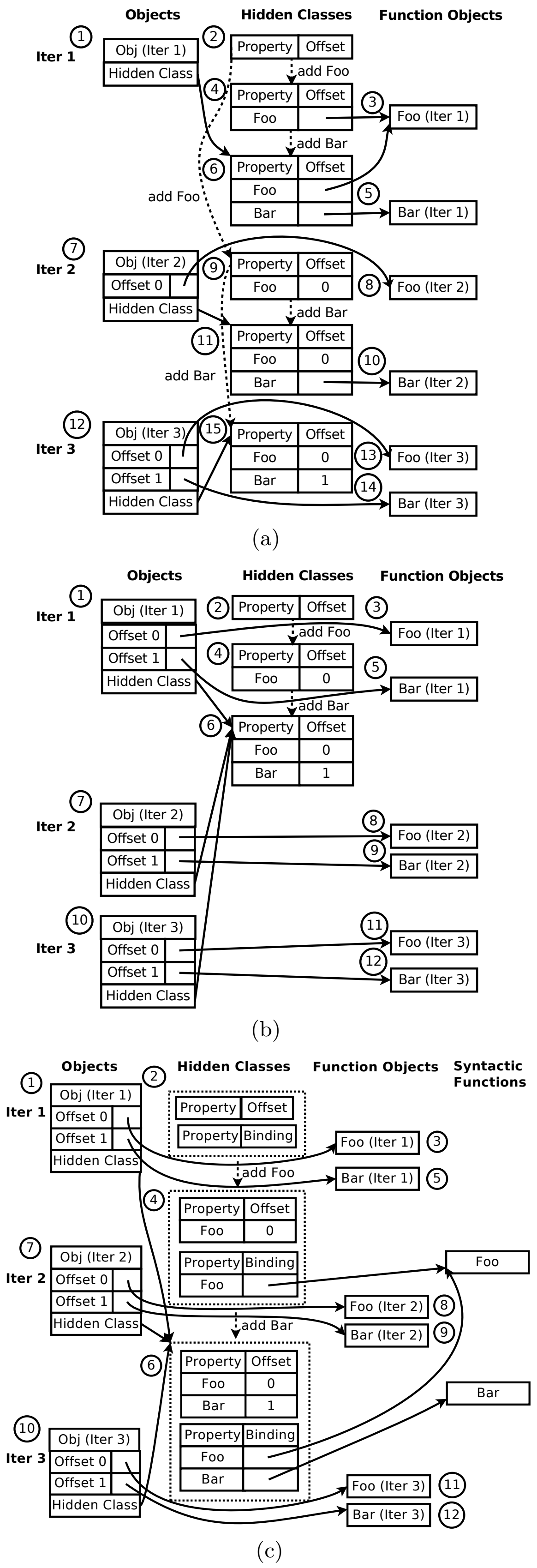

Figure 9: State of the heap after executing iterations of the loop in Figure 8: using the current V8 design (a), using our restructured V8 (b), and including syntactic function bindings within hidden classes (c).

function property Foo is assigned an anonymous function object (③), a new hidden class (④) is created with the new property. Notice that the pointer to the function object ③ is stored directly in the offset field of property Foo in hidden class ④, instead of in object ①. In line 4, for property Bar, a new function object (⑤) and a new hidden class (⑥) are created. In line 5, the call inline cache for Foo misses and records the type of Obj.

At iteration 2, a newly created object Obj (⑦) is initially typed to hidden class ②. However, when function property Foo is assigned in line 3, it *cannot reuse* hidden class ④. This is because the value of the hidden class function property, which points to function ③, differs from the pointer to the newly created function (⑧). At this point, V8 gives up on storing the method binding for Foo in the hidden class, and stores it in Obj (⑦) instead. The offset of the property is stored in a new hidden class (⑨). When the code reaches line 4 and tries to assign to property Bar, the property is not found in hidden class ⑨. So a new hidden class (⑪) is created that stores the binding from Bar to the new function object (⑩). At line 5, the call inline cache misses again because the type of Obj differs from iteration 1. Also, at this point, the inline cache turns polymorphic.

At iteration 3, a newly created object Obj (⑫) is again initially typed to hidden class ②. Also, when the function property Foo is assigned in line 3, the function object ⑬ is created, but no new hidden class is created because the type transitions to hidden class ⑨. When function property Bar is assigned in line 4, it cannot reuse hidden class ⑪ because the value of its function property differs from the newly created function (⑭). Hence, V8 gives up on storing the method binding for Bar in the hidden class and stores the value in Obj (⑫) instead. As before, the offset of the property is stored into a new hidden class (⑮). At line 5, the call inline cache misses yet again and the new type of Obj is recorded in the polymorphic cache.

For the rest of the iterations, all objects transition through hidden classes ②, ⑨, and ⑮. No new hidden classes are created and the type of Obj that reaches line 5 is always hidden class ⑮.

Through this example, note that, beyond the first inline cache miss, there are as many inline cache misses as there are function properties assigned to different instances of functions in the object. It is very common for websites to have large objects that include many function properties. When they exhibit this behavior, they decrease the type-hit-rate significantly. Also, note that even though all objects hit in the inline cache at line 5 starting from iteration 4, the inline cache will remain polymorphic for the rest of the iterations due to this design.

Finally, the real-world example of Figure 5(b) also suffers from the same problem due to the assignment of function properties addLogical and declareAvailable. Hence, for window.amznJQ to be truly monomorphic, we need to solve both the problems of prototypes and method bindings.

5. RESTRUCTURING THE TYPE SYSTEM

The previous section showed how changes in the prototypes or method bindings of otherwise identically-structured objects resulted in type unpredictability. In this section, we restructure the V8 compiler to decouple prototypes and method bindings from the type of an object, so a change in either does not result in the creation of a new hidden class.

5.1 Decoupling Prototypes From Types

We decouple prototypes from types by modifying internal data structures in the compiler such that the __proto__ pointer is moved from the hidden class to the object itself. With this change, individual objects now point directly to their respective prototypes. This change obviates the need to create a new hidden class whenever the prototype is changed. Below, we explain how this impacts the two problematic cases of function creation and built-in object creation, and briefly go over changes specific to each case.

5.1.1 Optimizing Function Creation

Figure 6(b) shows the state of the heap when executing the loop in Figure 5(a) after we restructure the compiler. At the first iteration, before the execution of Foo's body, an empty object Obj is created (① but empty) with its initial type set to the new hidden class ②, just as in Figure 6(a). The difference is that the __proto__ field that points to Foo.prototype (③) is now in object ① and not in its hidden class ②. After the execution of the body of Foo, hidden class ④ is created as the type of object Obj (①).

At the second iteration, object Obj (⑤) is created again with its initial type set to the hidden class ② that was created at the previous iteration. This is possible despite the different prototype object Foo.prototype (⑥) because __proto__ is no longer part of the hidden class. Similarly, after the execution of the body of Foo, the type of object Obj (⑤) is set to the hidden class ④ created at the previous iteration. Consequently, Obj now has the same hidden class at the end of each iteration. As a result, the store inline cache in line 3 of the example code always sees the same type and remains in monomorphic state.

In order to enable reuse of hidden classes across multiple dynamic instances of a function, the initial hidden class of a function (hidden class ② in the example) is created only once for each *syntactic function*. A syntactic function is a static function instance in the source code as given in the abstract syntax tree. The initial hidden class is cached in the internal syntactic function object shared by all instances of that function. A subsequent instance of the function uses that hidden class as the initial hidden class for its constructed objects.

5.1.2 Optimizing Built-in Object Creation

Extracting the __proto__ pointer from the hidden class enables reuse of the hidden classes from the previous global execution context. Specifically, objects that were created using the built-in functions that inherited from the built-in prototype objects can now reuse hidden classes across contexts. This is because the regeneration of prototype objects does not force the regeneration of hidden classes.

In order to enable reuse of hidden classes across multiple contexts, the initial hidden classes of built-in functions are cached across contexts as in the previous case. Once objects start from the same initial hidden class, they transition through the same hidden class tree that was created in previous executions. In this way, virtually all type unpredictability due to page loads can be eliminated.

5.2 Decoupling Method Bindings From Types

We propose two approaches to decoupling method bindings from types: complete and partial.

5.2.1 *Complete Decoupling*

We completely decouple method bindings from types by disallowing the storage of function property values in the hidden class altogether. Instead, function property values are always stored in the object itself; the hidden class only contains the offsets they are stored in, just like for regular properties.

Figure 9(b) shows the state of the heap at the end of each iteration of the loop in Figure 8 with Complete Decoupling. The function property values for `Foo` and `Bar` are directly stored in the allocated object already in the first iteration (①). Subsequent iterations transition through the same hidden classes (②, ④, ⑥). The reason is that these hidden classes store only property offsets which do not change across iterations. As a result, object `Obj` now has the same hidden class (⑥) at every iteration when it reaches line 5 of the example code. Hence, the call inline cache for `Foo` in line 5 always hits starting from iteration 2, and remains monomorphic throughout its lifetime.

5.2.2 *Partial Decoupling*

When method bindings are completely decoupled from types, inline cache hits on method calls require loading the method pointer from the object. This adds one extra load that slightly negatively impacts performance. With Partial Decoupling, we seek to eliminate this load by still keeping method bindings in types but in a way that does not result in excessive hidden class generation. Specifically, in the hidden classes, we store bindings to syntactic functions rather than to function objects. The binary code generated for a function is stored in the syntactic function object allocated for that function. Hence, only the binding to the syntactic function is needed for the purposes of calling a method.

Figure 9(c) shows the state of the heap after executing the loop in Figure 8 with Partial Decoupling. The dotted boxes denote hidden classes. Within a hidden class, there are two tables. The upper table stores the offsets for all the properties as before, while the lower one stores the syntactic function bindings of each method. Storing syntactic function bindings in hidden classes allows the compiler to generate efficient method calls. At the same time, not storing function object bindings in hidden classes removes all excessive type polymorphism. Note that the offsets of methods still need to be stored in the upper table, since the value of function properties can be loaded or stored just like any property, in which case, the function objects will be accessed.

We chose to implement Complete Decoupling for evaluation since we did not observe significant additional overhead due to indirect method calls for the benchmarks we tested. However, for codes with frequent method calls, Partial Decoupling may be more appropriate.

6. DISCUSSION

6.1 Impact of Our Compiler Modifications

Our enhancements to decouple prototypes and method bindings from object types target the dynamic JavaScript code often found in real websites. In the following, we consider how our modifications impact the performance and memory usage of such code, listing the advantages and the disadvantages.

6.1.1 *Impact on Performance*

Advantages. The most obvious advantage that comes from our modifications is that types are now much more predictable, resulting in increased type-hit-rate and less polymorphism in inline caches. Moreover, since optimizing compilers such as Crankshaft for V8 rely on type assumptions to hold inside the scope of optimization, type predictability can result in increased optimizations.

Disadvantages. Now that prototypes and method bindings are no longer immutable properties in the hidden class, the compiler cannot hard-code their values when generating code for inline caches. Instead, when accessing a prototype property or invoking a call to a method, we need to load the prototype pointer or the method pointer, respectively, from the object. This extra load can induce a slight delay even on inline cache hits, if a hit involves accessing a prototype property or calling a method.

6.1.2 *Impact on Memory Usage*

Advantages. The decoupling of prototypes decreases the total number of hidden classes in the heap, reducing memory pressure. For example, consider Figure 6. Let `I` be the number of iterations and `P` the number of properties assigned in the constructor. Then, the number of hidden classes before the decoupling is bound by `O(I * P)` while, after the decoupling, it is bound by only `O(P)`. The same occurs for the decoupling of method bindings. For example, consider Figure 9. Let `I` be the number of iterations and `P` the number of properties assigned using dynamically-allocated functions. Then, the number of hidden classes before the decoupling is bound by `O(min(I, P) * P)` while, after the decoupling, it is bound by only `O(P)`. Moreover, the reduced polymorphism in the inline caches also leads to a reduction in the amount of memory allocated to the code cache.

Disadvantages. When the `__proto__` pointer is moved from the hidden class to the object for prototype decoupling, it can cause increased memory use since, typically, there are more objects than hidden classes in the heap. The same can be said when function pointers are moved from the hidden class to the object for method binding decoupling. Also, the extra added pointers in the objects can require a tracing garbage collector to do extra traversals, slowing it down.

6.2 Other JavaScript Compilers

There exist popular JavaScript compilers other than V8, such as JavaScriptCore [3] for Safari, SpiderMonkey [6] for Firefox, and Chakra [1] for Internet Explorer. We have tried to find out if these compilers define types in the same way as V8. Unfortunately, this information is not publicly available, and one needs to dig into the compiler source code to find out — as we did for V8. Our initial research examining the JavaScriptCore source code seems to indicate that JavaScriptCore also creates hidden classes (called structures) and that, as in V8, the `__proto__` pointer in the hidden class is immutable. Also, past literature [14] seems to indicate that SpiderMonkey also includes prototypes as part of the type definitions of objects. Chakra is closed-source so we could not find out much.

Irrespective of how other compilers are implemented, we hope that our findings help JavaScript compiler writers and programmers understand the trade-offs and make the right design choices.

7. EVALUATION

7.1 Experimental Setup

To evaluate our enhancements, we use the JSBench benchmark suite [21]. JSBench assembles JavaScript code from real websites such as Amazon, Facebook, Google, Twitter, and Yahoo, by recording a series of user interactions on individual webpages. Each benchmark in JSBench records about one minute worth of user interactions on a single representative webpage from the website, starting from the page load. The benchmark models a common use of a website, where the user performs a few interactions with a certain webpage before reloading the page or getting redirected to another page. Since JavaScript execution differs slightly from browser to browser, JSBench sometimes pairs the same website with multiple browsers. We also briefly characterize the popular benchmark suites Kraken 1.1 [4], Octane [5], and SunSpider 0.9.1 [7], to see differences with JSBench.

We implement our enhancements in the most recent version of the Chrome V8 JavaScript compiler [8]. We build on top of the Full compiler, which is the lower-tier compiler of V8 that only does inline caching, and disable Crankshaft, which is the V8 optimizing compiler. As we saw in Section 3, Crankshaft does not improve JSBench in any way. Implementing our enhancements on the more complicated Crankshaft is more elaborate, and is left as future work.

To measure performance and memory management overhead, we run the benchmarks natively on a 2.4 GHz Xeon E5530 machine with an 8-MB last-level cache and a 24-GB memory. To measure other metrics, such as the dynamic instruction count and the number of inline cache accesses, we instrument the V8 runtime and the compiler-generated code with Pin [17]. To report the average of the benchmarks in a benchmark suite, we use the arithmetic mean (amean). The only exception is that we use the geometric mean (gmean) for the execution time.

In our experiments, for each benchmark, we execute three warm-up iterations before taking the actual measurements in the fourth iteration. The three warm-up iterations allow time for the code caches and inline caches to warm up. As indicated above, in the JSBench benchmarks, every iteration is preceeded by a page load; this is analogous to multiple visits to the same page during a single browser session.

We test four compiler configurations: *B*, *B**, *P*, and *C*. The baseline (*B*) is the original V8 compiler. *B** is the original V8 compiler after disabling the flushing of inline caches on page loads. This configuration is interesting because our enhancements (especially the built-in object optimization described in Section 5.1.2) rely on inline caches surviving page loads. The Chrome web browser flushes inline caches at every page load because, without our enhancements, inline caches are mostly useless across page loads. The rest of the configurations are built on top of *B**. Specifically, *P* is *B** enhanced with the *P*rototype decoupling of Section 5.1. Finally, *C* is *B** enhanced with the *C*ombination of both the prototype decoupling of Section 5.1 and the complete method binding decoupling of Section 5.2.1.

7.2 Characterization

We start by investigating why V8's optimizations have little impact on JSBench compared to the popular benchmarks, as seen in Figure 4. Figure 10 breaks down the dynamic instructions of the benchmarks in the *B* configuration. The benchmarks are organized as in Figure 4. Dynamic instructions are categorized into five types: *Code*, *Load_IC_Miss*, *Call_IC_Miss*, *Store_IC_Miss*, and *Runtime*. *Code* are instructions in the code generated by the V8 compiler. *Load_IC_Miss*, *Call_IC_Miss*, and *Store_IC_Miss* are instructions in the runtime inline cache miss handlers for loads, calls, and stores, respectively. *Runtime* are instructions in other runtime functions that implement JavaScript libraries such as JSON calls, string operations, and regular expressions.

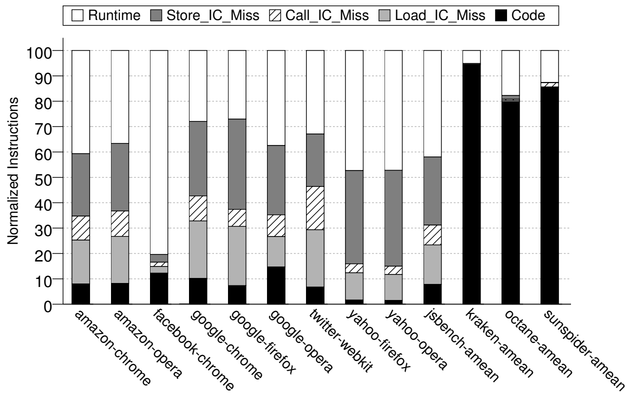

Figure 10: Breakdown of dynamic instructions by category.

The portion of instructions running compiler-generated code in JSBench is very small compared to the popular benchmarks. In JSBench, the bulk of the instructions execute in the runtime (including handling inline cache misses), and this is why compiler optimizations have so little effect. Also, on average, about half of all the instructions run inline cache miss handlers. These inline cache misses in JSBench are the target of our optimizations.

7.3 Impact on Performance

7.3.1 Execution Time

Figure 12 shows the execution time of JSBench for each of the four configurations normalized to *B*. We see that *B** is 23% slower than *B* on average, confirming that keeping inline caches across page loads does not benefit the baseline design. However, applying prototype decoupling (*P*) brings down the average execution time to 73% of *B*, and further applying method binding decoupling (*C*) brings it down to 64% of *B*. In theory, method binding decoupling can be applied without prototype decoupling; however, it only improves performance marginally if applied by itself. Overall, our optimizations reduce the execution time of JSBench by 36% on average.

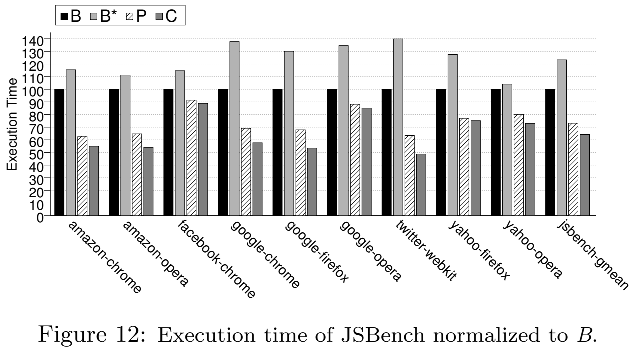

Figure 12: Execution time of JSBench normalized to *B*.

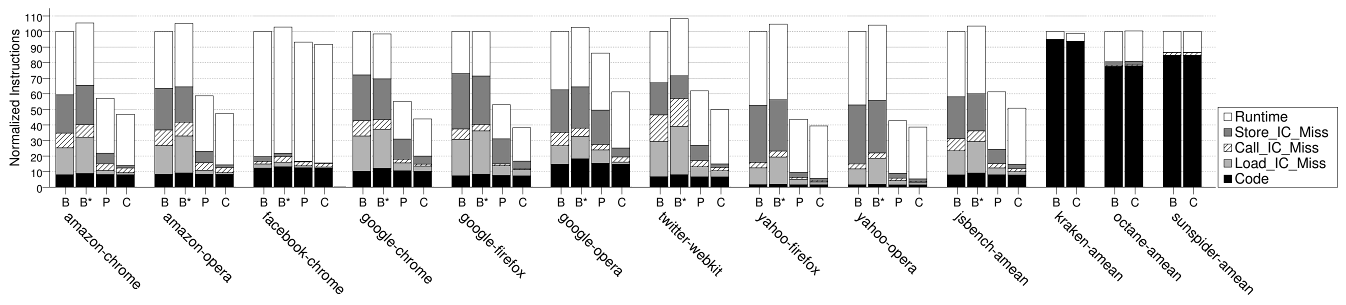

Figure 11: Dynamic instruction count normalized to the original V8 compiler (B).

7.3.2 Type Predictability

We now examine the improvement in type predictability due to our enhancements. We assess the improvement by measuring increases in the type-hit-rate and reductions in polymorphism for all inline caches. Figure 13 shows a breakdown of inline cache accesses into five categories: *Initial Misses*, *Polymorphic Misses*, *Monomorphic Misses*, *Polymorphic Hits*, and *Monomorphic Hits*. *Initial Misses* are inline cache misses that happen when an inline cache is accessed for the first time and there is no previously recorded type. *Polymorphic Misses* and *Monomorphic Misses* are misses suffered by an inline cache when it is in the polymorphic or monomorphic state, respectively. *Polymorphic Hits* and *Monomorphic Hits* are the same for inline cache hits.

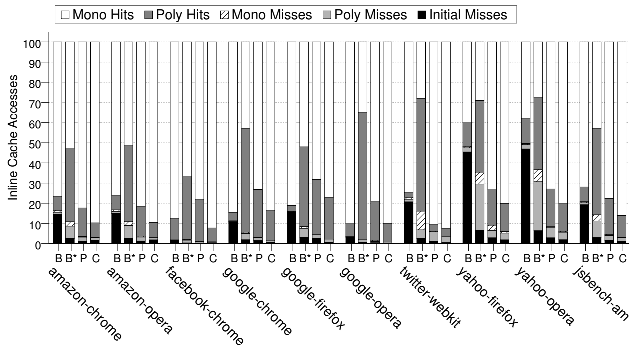

Figure 13: Breakdown of inline cache accesses by category.

B suffers significantly from *Initial Misses* due to the flushing of inline caches at every page load. B^* suffers much less from *Initial Misses*, as inline caches are allowed to warm up. However, without our enhancements, many accesses miss anyway and become *Polymorphic Misses* and *Monomorphic Misses*. Also, the number of *Polymorphic Hits* increases greatly, as inline caches accumulate different types from previous warm-ups. However, with P and then C, most of the misses and much of the polymorphism go away. The result is that C has a 97% overall hit rate, and an 86% monomorphic hit rate. In contrast, B has a 79% overall hit rate and a 72% monomorphic hit rate.

7.3.3 Dynamic Instruction Count

Figure 11 shows the dynamic instruction count of JSBench for all the configurations. There are also bars for the averages for Kraken, Octane, and SunSpider for the B and C configurations. All bars are normalized to B and broken down into the same categories as in Figure 10. Kraken, Octane, and SunSpider show no improvements as expected, but we note that our modifications do not cause overhead either. Overheads could come from the extra loads required for prototype property accesses and method calls, as explained in Section 6.1.1. Also, the fact that the number of instructions spent in *Code* is almost identical between C and B in JSBench shows that these overheads are negligible in JSBench too. The rest of this section will focus exclusively on JSBench.

For JSBench, the average number of instructions increases slightly to 104% in B^* but decreases to 61% in P and decreases further to 51% in C. Overall, our optimizations eliminate on average 49% of the dynamic instructions in JSBench.

Looking at the instruction breakdown, we see that the reduction in dynamic instructions comes from a lower inline cache miss handling overhead, which in turn comes from improvements in type predictability enabled by our optimizations. With all of our enhancements applied (C), inline cache miss handling accounts for only 13% of the dynamic instructions, compared to 50% in B. The sources of the remaining misses in C are non-type-related. They include cases such as loads from non-existent properties of an object, or accesses to special properties defined by "getters" and "setters", which require callbacks to user-defined functions.

Comparing B to B^*, the increase in dynamic instructions is often due to a higher number of instructions spent handling inline cache misses in B^*. This is despite the consistent decrease in the number of inline cache misses when going from B to B^* in Figure 13. The reason is because misses in B^* are more expensive on average. The V8 compiler has a policy of not generating code for the very first miss of an inline cache. The purpose is to refrain from performing expensive code generation for inline caches that will only be accessed once. This policy makes *Initial Misses* much cheaper than *Monomorphic Misses* or *Polymorphic Misses*, which are prevalent in B^*.

Finally, the increase in instructions from B to B^* translates into a large increase in execution time in Figure 12. We believe this is due to branch misprediction and cache effects from polymorphism, although we did not investigate further.

7.3.4 Function Property Binding Optimization

Section 4.2 described how inline cache misses resulting from object type polymorphism due to method bindings can go away after enough iterations of the code. Figure 14 shows how many warm-up iterations are needed for this to happen with JSBench. The bars in the figure show dynamic instructions broken down into the usual five categories. There are bars for P and C, which perform 3 warm-ups as usual. The bars labeled *10*, *20*, and *50* correspond to P after the given number of warm-ups. For a given benchmark, the bars are

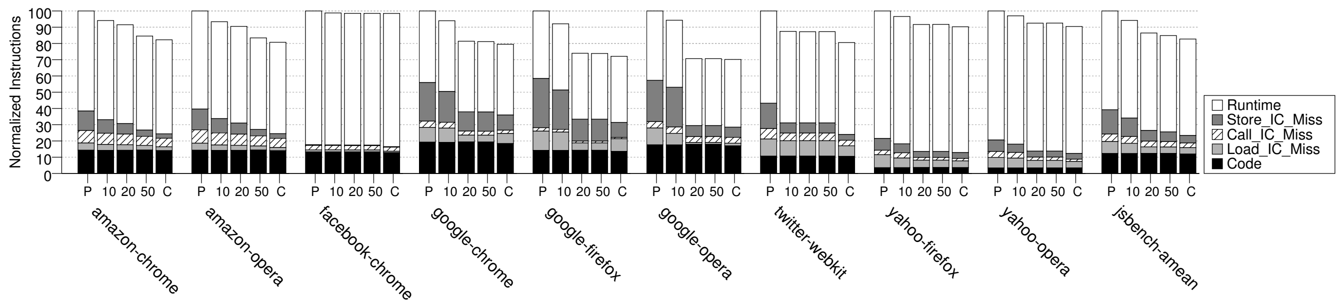

Figure 14: Dynamic instruction count when running with configuration *P* after 3, 10, 20, and 50 warm-up iterations, and when running with configuration *C*.

Benchmark	Heap Allocated								Allocated Heap Reduction (%)	Collection Time Increase (%)
	Baseline				Enhanced					
	Object (KB)	Code (KB)	Hidden Class (KB)	Total (KB)	Object (KB)	Code (KB)	Hidden Class (KB)	Total (KB)		
amazon-chrome	208	86	110 (843)	404	217	37	13 (128)	267	33.9	4.7
amazon-opera	205	88	113 (848)	406	211	34	13 (129)	258	36.5	5.9
facebook-chrome	6353	150	741 (3156)	7244	6401	173	48 (482)	6622	8.6	1.3
google-chrome	1229	410	387 (3228)	2026	1295	232	53 (439)	1580	22.0	2.9
google-firefox	925	295	346 (2998)	1566	970	169	40 (326)	1179	24.7	1.9
google-opera	5542	786	691 (4944)	7019	5955	472	227 (2099)	6654	5.2	5.3
twitter-webkit	177	37	46 (316)	260	185	10	0 (15)	195	25.0	6.2
yahoo-firefox	339	149	123 (1034)	611	356	136	46 (365)	538	11.9	4.8
yahoo-opera	342	153	127 (1069)	622	360	143	50 (405)	553	11.1	5.3
jsbench-amean	1702	239	298 (2048)	2240	1772	156	54 (488)	1983	19.9	4.3
kraken-amean	468496	0	0 (0)	468496	468837	0	0 (0)	468837	-0.1	0.1
octane-amean	37405	1	0 (0)	37406	37427	1	0 (0)	37428	-0.1	-3.4
sunspider-amean	3194	0	0 (0)	3194	3222	0	0 (0)	3222	-0.9	12.2
bench-amean	169698	0	0 (0)	169698	169829	0	0 (0)	169829	-0.4	3.0

Table 1: Heap memory allocated and heap collection time for JSBench (top) and the other benchmark suites (bottom).

normalized to *P*.

The results show that most websites need tens of page loads for *P* to approach *C* without the method binding decoupling enhancement. Some websites such as *twitter-webkit* perform significantly worse than *C* even after 50 page loads.

7.4 Impact on Memory Management

In this section, we assess how our optimizations impact two aspects of memory management: allocated heap memory and garbage collection overhead. Consider the heap memory first. For each JSBench website, we measure the new heap memory allocated during the measured iteration. Recall that we measure one iteration of the website execution after three warm-up iterations, where each iteration includes a page reload. With our optimizations, the measured iteration allocates less heap memory than before for two reasons. First, the iteration is able to substantially reuse inline cache code and hidden classes from previous iterations. Second, the iteration is also able to reduce the number of hidden classes used even within an iteration.

The upper part of Table 1 shows the new heap allocated in the measured iteration for each JSBench website. Columns 2-5 correspond to the baseline compiler, while Columns 6-9 correspond to the compiler with our enhancements. In each case, we show the heap consumed by JavaScript objects, compiler-generated code, hidden classes, and the total. For hidden classes, the number inside the parentheses is the number of hidden classes generated. Looking at the JSBench average row, we see that our optimizations increase the object memory (due to adding the `__proto__` pointer to objects). However, thanks to the two reasons described above, the code memory and, especially, the hidden class memory, go down. The result is a lower total heap allocation. Column 10 shows the total reduction in heap memory allocated in each website due to our optimizations relative to the baseline. On average, we reduce the heap memory allocated by a sizable 19.9%. This will make garbage collections less frequent.

For comparison, the lower part of Table 1 shows data for the three popular benchmark suites. Recall that, for these benchmarks, we also measure one iteration after three warm-up iterations, and there are no page reloads. We can see that there is barely any generation of new hidden classes or code after warm-up. This is expected, since types are very stable in these benchmarks. In addition, it can be shown that most objects in these benchmarks are primitives such as integers and floats that do not have prototypes. As a result, the added memory due to the extra `__proto__` pointer in objects is minimal. Consequently, our optimizations barely change the heap allocation. As shown in Column 10, our optimizations increase the total heap allocated by 0.4% on average for the three benchmark suites.

We now consider the overhead of collecting the heap used in the measured iteration of JSBench. Before we execute the iteration, we run the garbage collector. Then, after we execute the iteration, we immediately trigger the collection of all the heap used in the iteration. For the compiler with our optimizations, we collect both the new heap generated in the measured iteration plus the heap reused from previous iterations; for the baseline compiler, we only need to collect the new heap generated in the measured iteration, since there is no reuse from previous iterations.

In our enhanced compiler, the time to collect this heap in JSBench may be higher or lower than in the baseline compiler. On the one hand, it may be higher because, by moving the `__proto__` pointers from the hidden classes to objects, we require the V8 mark-and-sweep collector to traverse more pointers — in the case where there are multiple objects of each type. On the other hand, the time may be lower because our optimizations reduce the number of hidden classes, even without reuse across iterations.

The last column of Table 1 shows the resulting increase in heap collection time with our optimizations relative to the baseline. We see that, on average for the JSBench websites, we increase the time by 4.3%. This means that the first effect listed above dominates without reuse. Given that the new heap generated by the enhanced compiler is 19.9% smaller and most of the reduction comes from reuse, we expect collection time to decrease if we were to collect only the *new* heap. But unfortunately collecting only the new heap is infeasible. For the three popular benchmark suites, the time increases by a modest 3.0% on average. This is consistent with the relatively small increase in total heap memory use.

8. RELATED WORK

Richards et al. [23, 22] and Ratanaworabhan et al. [20] have performed empirical studies to evaluate how programmers use some of the JavaScript features in real websites, as well as in some of the popular benchmark suites. These studies have shown that the popular benchmarks are different from the websites with respect to call site dynamism, the frequency of property additions and deletions after object construction, constructor polymorphism, and how programmers use `eval`. These works were done at the program behavioral level, and did not analyze how these behaviors impact the compiler and the resulting performance.

Modern JavaScript engines use key ideas from SmallTalk [13] and Self [11] on how to optimize dynamically-typed, object oriented languages, such as hidden classes [11], inline caches [13], and polymorphic inline caching [15].

There are other proposed techniques to speedup JavaScript programs that are orthogonal to the techniques discussed in this paper; they could be used together with our compiler modifications. For instance, recent works decrease overheads through program analysis, to statically infer a variable's type [14, 16]. Another proposal extends the ISA to load and check the type with a single instruction [9]. Finally, ParaGuard [19] and ParaScript [18] propose hardware support for parallel execution of JavaScript programs. The former offloads the runtime checks to another thread while the main thread continues with the execution of the user code; the latter uses speculative parallelization to support loop-level parallelization of JavaScript programs.

9. CONCLUSIONS

This paper analyzed the impact of the Chrome V8 compiler optimizations on JavaScript code from real websites assembled by JSBench, and found that it lags far behind the impact on popular benchmarks. We identified the core problem hampering optimizations as type unpredictability. The problem stems from the way the compiler understands the notion of types. V8 encodes into types two pieces of information unrelated to object structure: (1) the inherited prototype and (2) method bindings. This was done assuming that the behavior of JavaScript code mimics that of statically-typed languages, where the inherited class and method bindings cannot change once an object is created. We showed that this assumption is often false for JavaScript code used in real websites.

We proposed three optimizations that, by rethinking types to accommodate the dynamic behavior of JavaScript website code, eliminate most type unpredictability. In JSBench, these optimizations reduced, on average, the execution time by 36%, and the dynamic instruction count by 49%. Moreover, the heap memory allocated decreased by 20%, due to reductions in hidden classes and inline cache code.

10. REFERENCES

[1] Chakra. http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx.

[2] ECMAScript. http://www.ecmascript.org/.

[3] JavaScriptCore. http://trac.webkit.org/wiki/JavaScriptCore.

[4] Kraken Benchmarks. http://krakenbenchmark.mozilla.org/.

[5] Octane Benchmarks. https://developers.google.com/octane.

[6] SpiderMonkey Project. https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[7] SunSpider Benchmarks. http://www.webkit.org/perf/sunspider/sunspider.html.

[8] V8 JavaScript Engine. https://developers.google.com/v8/.

[9] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers. Checked load: Architectural support for JavaScript type-checking on mobile processors. In *HPCA*, 2011.

[10] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *OOPSLA*, 2012.

[11] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *OOPSLA*, 1989.

[12] D. Clifford. Breaking the JavaScript limit with V8. http://v8-io12.appspot.com.

[13] L. P. Deutsch and A. M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *POPL*, 1984.

[14] B. Hackett and S.-Y. Guo. Fast and precise hybrid type inference for JavaScript. In *PLDI*, 2012.

[15] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP*, 1991.

[16] S. Li, B. Cheng, and X.-F. Li. TypeCastor: Demystify dynamic typing of JavaScript applications. In *HiPEAC*, 2011.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[18] M. Mehrara, P.-C. Hsu, M. Samadi, and S. Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *HPCA*, 2011.

[19] M. Mehrara and S. Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *CGO*, 2011.

[20] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *the USENIX Conference on Web Application Development*, 2010.

[21] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, 2011.

[22] G. Richards, C. Hammer, B. Burg, and J. Vitek. The Eval that men do: A large-scale study of the use of Eval in JavaScript applications. In *ECOOP*, 2011.

[23] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *PLDI*, 2010.