Improving JavaScript Performance by Deconstructing the Type System[*]

Wonsun Ahn
University of Pittsburgh
{wahn}@pitt.edu

Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas
University of Illinois at Urbana-Champaign
{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 dynamic instruction count of JSBench by 49%.

1. MOTIVATION

Figure 1 compares the number of instructions executed by several JavaScript benchmarks with different V8 optimization levels. We show data for the JSBench [5], Kraken [1], Octane [2], and SunSpider [3] 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. 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 traditional compiler optimizations. Finally, *No Crankshaft, No IC* is *No Crankshaft* with inline caching disabled.

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 negligible effect on the JSBench suite. The goal of this work is to explain why real websites as assembled in JSBench do not benefit from neither inline caching or V8 optimizations, and propose solutions to the problem.

[*]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).

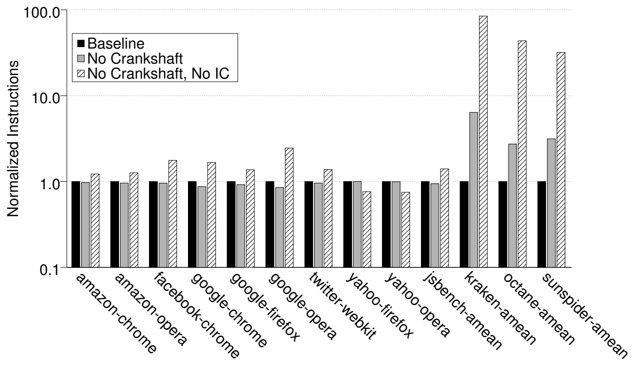

Figure 1: Comparing the number of instructions executed with different V8 optimization levels.

2. 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 after a simple type check. Also, a type miss sometimes requires the creation of a new hidden class for the new type — 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 in an inline cache.

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.

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.* 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 *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. This is done by integrating the prototype and method binding information into hidden classes. Since hidden classes are immutable, this allows a single check of the hidden class in inline caches to suffice as a check for, not only the set of properties, but also the prototype and method bindings. This allows the compiler to generate optimized code when traversing the prototype inheritance chain or when performing method calls, in the same way it can optimize for property accesses. However, the downside is that, to maintain the immutability of hidden class, a *new hidden class* needs to be created for every modification.

For the popular JavaScript benchmarks that compiler developers compete on, the assumption that prototypes and method bindings rarely 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 1.

Figure 2(a) shows an example where prototype changes can lead to type unpredictability. A loop creates a new function object at each iteration, assigning it to the variable `Foo`, and along with it its associated prototype object. Also at each iteration, a new object `Obj` is constructed using that function. Since a new object in JavaScript always inherits from the prototype associated with its constructor, `Obj` ends up having a different prototype, and hence a new hidden class, at each iteration. 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`.

Figure 2(b) shows an example where method binding changes can lead to type unpredictability. A loop 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. In reality, the type of `Obj` that reaches line 5 changes across iterations due to different method bindings at `Foo` and `Bar`.

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

(b)

Figure 2: Code patterns that lead to high type unpredictability due to (a) prototype changes and (b) method binding changes.

3. RESTRUCTURING THE TYPE SYSTEM

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.

We decouple prototypes from types by modifying internal data structures in the compiler such that the `__proto__` pointer (the pointer to the parent prototype) 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.

With prototype decoupling, all instances of `Obj` in Figure 2(a) can now share a single hidden class. In order to enable reuse of a single hidden class across multiple dynamic instances of a function (`Foo` in the example), we modify the compiler such that now all objects created by the same *syntactic function* start from the same initial hidden class. 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 with the creation of the first instance. A subsequent instance of the function uses that hidden class as the initial hidden class for its constructed objects.

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

With Complete Decoupling, we entirely decouple method bindings from types by disallowing the storage of method bindings in the hidden class altogether. Instead, method bindings are always stored in the object itself in the form of pointers to function objects. Now, this can result in slower method calls, since the compiler can no longer optimize the calls based on the bindings stored in the immutable hidden class.

With Partial Decoupling, we seek to keep method bindings in hidden classes to still optimize calls but in a way that does not result in excessive hidden class generation. Specifically, we still store the function object pointers in the object itself such that updates to them don't cause new hidden class creation. But, in addition, we also store bindings to syntactic functions in the hidden classes to optimize calls. 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.

4. EVALUATION

We implement our enhancements in the Chrome V8 JavaScript compiler [4]. We build on top of the Full compiler, which is the lower-tier compiler of V8 that only does inline caching, and disable Crankshaft. As we saw in Section 1, 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.

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. 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**. Specif-

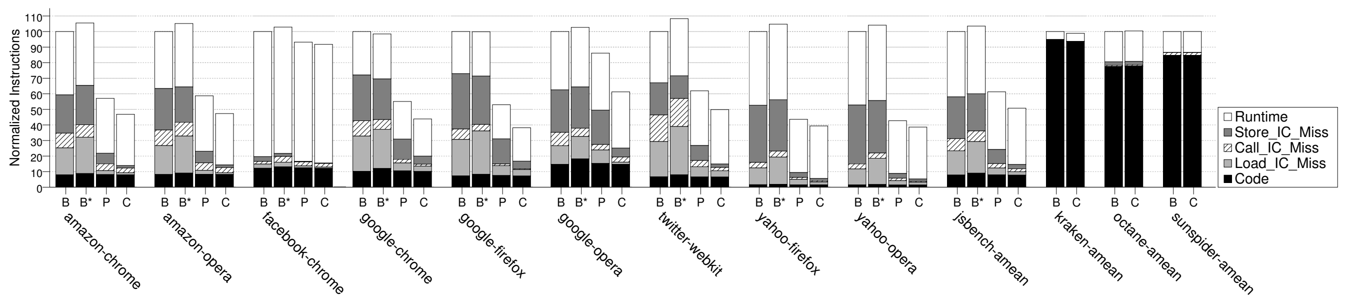

Figure 3: Dynamic instruction count normalized to the original V8 compiler (B).

ically, P is $B*$ enhanced with Prototype decoupling. Finally, C is $B*$ enhanced with the Combination of both prototype decoupling and method binding decoupling.

Figure 3 shows the dynamic instruction counts for the given configurations when executing the various benchmarks. All bars are normalized to B. 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.

Kraken, Octane, and SunSpider show no improvements as expected, but we note that our modifications do not cause overhead either. For JSBench, our optimizations eliminate on average 49% of the dynamic instructions in JSBench when all enhancements are applied (C). 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.

In terms of heap memory footprint, our optimizations has the potential to increase heap memory for objects due to adding the `__proto__` pointer to objects. On the other hand, the heap memory dedicated to inine cache code and the hidden class metadata can go down due to the reduction of hidden classes and inline cache misses. On average, we actually reduce the total heap memory allocated by a sizable 20% for JSBench. The other three benchmark suites had a negligible increase of 0.4%.

5. 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 rethinking types to accommodate the dynamic behavior of JavaScript website code, eliminating most type unpredictability. In JSBench, these optimizations reduced, on average, the the dynamic instruction count by 49% and heap memory allocated by 20%.

6. REFERENCES

[1] Kraken Benchmarks. http://krakenbenchmark.mozilla.org/.
[2] Octane Benchmarks. https://developers.google.com/octane.
[3] SunSpider Benchmarks. http://www.webkit.org/perf/sunspider/sunspider.html.
[4] V8 JavaScript Engine. https://developers.google.com/v8/.
[5] G. Richards, A. Gal, B. Eich, and J. Vitek. Automated construction of JavaScript benchmarks. In *OOPSLA*, 2011.