Super-Bytecode Construction for Accelerating JavaScript Object Property Accesses

Seong-Won Lee, Soo-Mook Moon
Seoul National University

ABSTRACT

JavaScript execution during web page loading spends much of its time for executing runtime services of the JavaScript engine, especially for accessing properties of objects. One problem is that many of these object properties are accessed only once during web page loading. This makes the bytecode for these one-time accesses be executed without optimizations such as inline caching, while suffering from the overhead of creating property maps for each access. This paper proposes *super-bytecode* for merging a sequence of property accesses for the same object, with optimized runtime services to accelerate them. It also improves inline caching. Our preliminary experimental results show that the super-bytecode accelerates the loading of some web pages.

1. Introduction

Web pages are programmed using HTML5, CSS, and JavaScript. During web page loading, the browser parses the HTML document and builds a document object model (DOM) tree, which is then displayed on the screen based on CSS by the rendering engine. When the JavaScript tag is met during the HTML parsing, the corresponding JavaScript code is executed by the JavaScript engine, mostly for initializing objects and registering event handlers. JavaScript code is first parsed to the bytecode, which is either interpreted or translated to machine code by the just-in-time compiler (JITC) for faster execution. The JavaScript execution takes a significant (34%) portion of the web page loading time [2].

We found that JITC is not effective for web pages, especially for accelerating web page loading. For example, Figure 1 shows the performance of the JITC for the JavaScript execution time during the loading of some web pages, compared to that of the interpreter, when we experiment with the WebKit JavaScriptCore (JSC) engine [5] (version 1.4.0); they are (1) abc.com, (2) digg.com, (3) iht.com, (4) maxim.com, (5) nationalgeographic.com, (6) nydailynews.com, (7) reuters.com, (8) slashdot.org, (9) wsj.com, and (10) Sunspider benchmark for comparison, respectively. For most web pages except for (8), the JITC shows a worse performance than the interpreter. This is in sharp contrast for the Sunspider benchmark where the JITC is much better. Actually, when we turn on JITC for any latest JavaScript engines (e.g., DFG JITC or V8 Crankshaft), we rarely see an improvement, but only a degradation for web pages or apps.

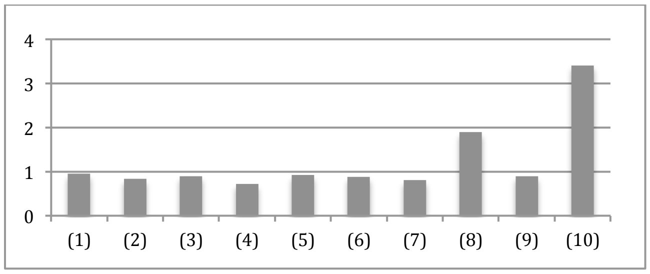

Figure 1. JavaScript execution time during web page loading.

One reason is that JavaScript execution during web page loading spends much of its time for executing *runtime services* or the *native functions* of the JavaScript engine, which cannot be accelerated by the JITC. Runtime services are for executing some complex jobs such as object property accesses or floating point operations, requested by the interpreter or the JITC. They are part of the functions in the JavaScript engine in C++. Native functions are for JavaScript built-in functions (e.g., math) or for API functions (e.g., DOM API) in C++. When we experiment with JITC enabled, Figure 2 shows the distribution of JavaScript running time during the web page loading. Runtime services and native function take more than 60% of the running time. Parsing and JITC overhead is also significant, leaving the runtime portion of the JITC-generated machine code somewhat marginal.

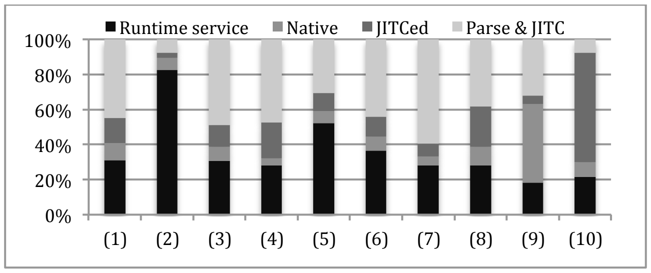

Figure 2. Distribution of JavaScript execution during loading.

The above graph indicates the JITC is not effective for web page loading. On the other hand, there is not much to accelerate the interpreter routine itself. And, the native functions are fixed, so there is nothing to accelerate them, either. The only thing left to optimize is the runtime services of the JavaScript engine. We found that the runtime services for object property accesses are dominant among the execution time of the runtime services as shown in Figure 3. We also found that 84% of the accesses are the one-time accesses, and repeated accesses are just 16%.

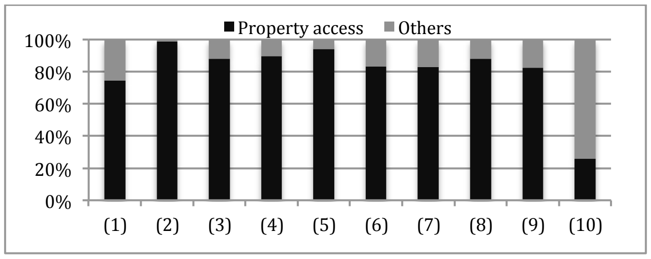

Figure 3. Portion of property access in runtime services.

Based on these observation, we attempt to optimize the object property accesses during the web page loading, in the context of interpreter, especially for the first-time accesses, most of which are one-time accesses. We first explain the overhead involved with the object property access, especially the first-time access.

A JavaScript object has a *storage table* which has the values of its properties and a *property map* which describes the offset of each property in the storage table [3]. For example, consider a function *point()* in Figure 4 (a), which is used as a constructor of an object to initialize its properties. Figure 4 (b) shows the bytecode generated by the parser of the JSC [5], where *put_by_id* will initialize each property. The runtime service for *put_by_id* will first check if the property exists in the object; if not, it will add the property in the storage table and create a new property map based on the old one, added with the offset of the new property. So, each *put_by_id* will generate a new property map as in Figure 4 (c). If *point()* is called again to create a new object, the previous property maps will be reused without creating them again, though [3]. However, web-page loading is involved with many one-time object property accesses, so the creation of property maps for each bytecode can be an overhead.

Another overhead of object accesses is related to *inline caching* [4]. If *point()* is called again repetitively as in Figure 4 (a) and the object structure (i.e., the constructor *point()*) does not change over iterations, it would be better to remember the offset and use it directly instead of accessing the property map. So, the bytecode *put_by_id* is replaced by a quicker version where the offset and the address of the property map are saved. When it is interpreted, the address is compared to the current address of the property map first, and if they are the same, the offset in the bytecode is used directly to access the property in the storage table without using the runtime services. This is called *inline caching*. The issue is that we need to compare the address three times for each access in Figure 4 (a), for example, which we want to reduce.

Our optimization approach is using a *super-bytecode* instruction which merges multiple object-access bytecodes into one bytecode. This can reduce the overhead of the runtime services of individual bytecode, especially for the first-time object accesses. For inline caching, super-bytecode can also reduce the overhead of comparing multiple addresses of property maps by making a single comparison with the final property map. We will describe how to generate the super-bytecode in the next section

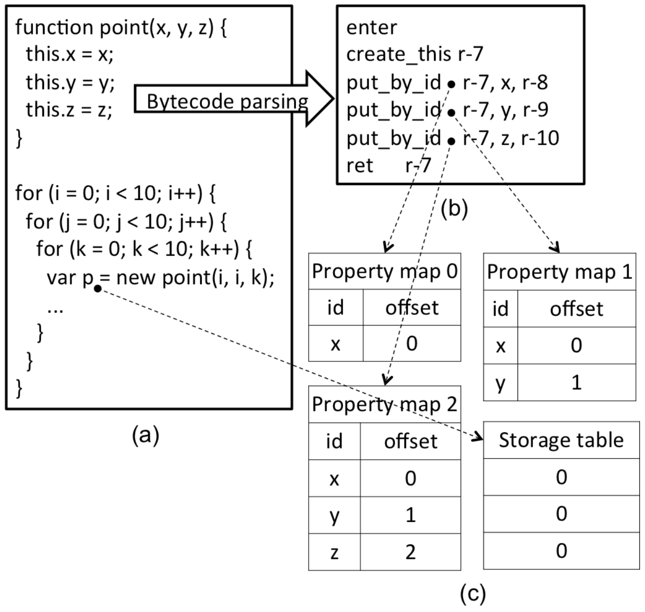

Figure 4. An example of the object property accesses.

2. Super-Bytecode Construction (SBC)

When a function is called for the first time, we perform the super-bytecode construction (SBC) for the method. To find the candidate object accesses for SBC, we first perform an analysis on the basic block (BB) boundary to reduce the analysis overhead. In the current implementation, we simply merge a sequence of *put_by_id*s for the same object in the BB into a single bytecode *m_put_by_id*. Similarly, a sequence of *get_by_id*s for the same object is replaced by a single bytecode *m_get_by_id*. In Figure 4 (b), we can replace the three *put_by_id*s which write the three properties for the same object by a new bytecode *m_put_by_id r-7, {x,y,z}, {r-8,r-9,r-10}*. This will write the three properties by a merged, optimized runtime service routine, instead of three runtime service routines.

The benefit of the super-bytecode is two-folds. We can reduce the overhead of creating the property map for each individual bytecode for the first-time object property accesses. In Figure 4 (c), it is an overhead to create the property map0 and map1 since only the property map2 is enough to resolve the offset of x, y, and z. Even when the function *point()* is called repetitively, the property map2 is enough to resolve the offsets. Our super-bytecode will obviate the overhead of creating the extra property maps or accessing them, which will be more useful as more property accesses are merged, even if they are executed only once.

We can also have a benefit when the super-bytecode is executed repetitively. Inline caching requires the address comparison of the property map, which we need to do for each access. In Figure 4 (c), for example, we need to compare three times for the three accesses with the corresponding property maps. SBC requires saving only one address and comparing only once, which would accelerate the inline-cached, property accesses.

3. Experimental Result

We implemented the proposed SBC for the JSC interpreter in a WebKit-based web browser (the JIT compiler is disabled). We measured the JavaScript execution time during the loading of web pages and the running of the Sunspider benchmark on an x86-based environment. Figure 5 shows the interpretation time speed-up with SBC over the interpretation with the original bytecode. We can find tangible improvement for some web pages. Also at least, there is no serious slow-down from the optimization overhead for the other web pages and the benchmark. We are still working on further optimizations based on this promising result.

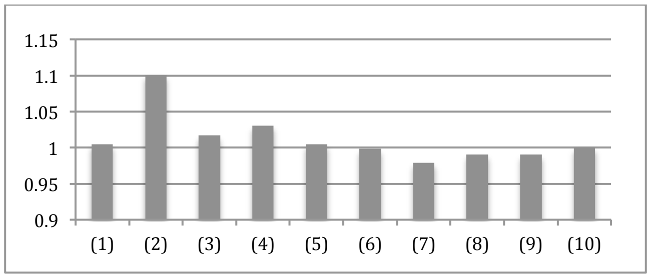

Figure 5. Speed-up of super-bytecode compared to original.

4. References

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

[2] S. Lee and S. Moon. Selective Just-in-Time Compilation for Client-side Mobile JavaScript Engine. In CASES, 2011

[3] C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based On Prototypes. In OOPSLA, 1989.

[4] U. Holzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches. In ECOOP, 1991.

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