Computer Languages, Systems & Structures 40 (2014) 37–52

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Just-in-time value specialization

Igor Rafael de Assis Costa, Henrique Nazaré Santos, Péricles Rafael Alves, Fernando Magno Quintão Pereira*

Department of Computer Science, Federal University of Minas Gerais (UFMG), Brazil

ARTICLE INFO

Article history:
Received 8 February 2013
Received in revised form
9 October 2013
Accepted 1 November 2013
Available online 13 November 2013

Keywords:
Just-in-time compilation
JavaScript
Speculation

ABSTRACT

JavaScript emerges today as one of the most important programming languages for the development of client-side web applications. Therefore, it is essential that browsers be able to execute JavaScript programs efficiently. However, the dynamic nature of this programming language makes it very challenging to achieve this much needed efficiency. In this paper we propose parameter-based value specialization as a way to improve the quality of the code produced by JIT engines. We have empirically observed that almost 60% of the JavaScript functions found in the world's 100 most popular websites are called only once, or are called with the same parameters. Capitalizing on this observation, we adapt a number of classic compiler optimizations to specialize code based on the runtime values of function's actual parameters. We have implemented the techniques proposed in this paper in IonMonkey, an industrial quality JavaScript JIT compiler developed at the Mozilla Foundation. Our experiments, run across three popular JavaScript benchmarks, SunSpider, V8 and Kraken, show that, in spite of its highly speculative nature, our optimization pays for itself. As an example, we have been able to speed up V8 by 4.83%, and to reduce the size of its generated native code by 18.84%.

1. Introduction

JavaScript is presently the most important programming language used in the development of client-side web applications [1]. If in the past only very simple programs would be written in this language, today the reality is different. JavaScript is ubiquitously employed in programs ranging from simple form validation routines to applications as complex as Google's Gmail. Furthermore, JavaScript is also used as the target intermediate representation of frameworks such as the Google Web Toolkit. Thus, it fills the role of an assembly language of the Internet. Given that every browser of notice has a way to run JavaScript programs, it is not surprising that industry and academia put considerable effort in the creation of efficient execution environments for this programming language.

However, executing JavaScript programs efficiently is not an easy task. JavaScript is a very dynamic programming language: it is dynamically typed, it provides an `eval` function that loads and runs strings as code, and its programs tend to use the heap heavily. This dynamic nature makes it very difficult for a static compiler to predict how a JavaScript program will behave at runtime. In addition to these difficulties, JavaScript programs are usually distributed in source code format, to ensure portability across different computer architectures. Thus, compilation time generally has impact on the user experience. Today, the just-in-time compiler seems to be the tool of choice of engineers to face all these challenges.

* Corresponding author. Tel.: +55 31 3409 5860; fax: +55 31 3409 5858.
E-mail address: fernando@dcc.ufmg.br (F. Magno Quintão Pereira).

1477-8424/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.cl.2013.11.001

A just-in-time compiler either compiles a JavaScript function immediately before it is invoked, as Google's V8 does, or while it is being interpreted, as Mozilla's TraceMonkey did. The advent of the so-called *Browser War* between main software companies has boosted significantly the quality of these just-in-time compilers. In recent years we have seen the deployment of very efficient trace compilers [2–4] and type specializers for JavaScript [5]. New optimizations have been proposed to speed up JavaScript programs [6,7], and old techniques [8] have been reused in state-of-the-art browsers such as Google Chrome. Nevertheless, we believe that the landscape of current JIT techniques still offers room for improvement, and our opinion is that much can be done in terms of runtime value specialization.

As we show in Section 2, we have observed empirically that almost 60% of all the JavaScript functions in popular websites are either called only once, or are always called with the same parameters. Similar numbers can be extended to typical benchmarks, such as V8, SunSpider and Kraken. Grounded by this observation, in this paper we propose to use the runtime values of the actual parameters of a function to specialize the code that we generate for it.[1] In Section 3 we revisit a small collection of classic compiler optimizations under the light of the proposed approach. As we show in the rest of this paper, some of these optimizations, such as constant propagation and dead-code elimination, perform very well once the values of the parameters are known. This knowledge is an asset that no static compiler can use, and, to the best of our knowledge, no just-in-time compiler currently uses.

We have implemented the ideas discussed in this paper in IonMonkey, a JavaScript JIT compiler that runs on top of the SpiderMonkey interpreter used in the Firefox browser. As we explain in Section 4, we have tested our implementation on three popular JavaScript benchmarks: V8, SunSpider and Kraken. We only specialize functions that are called with at most one different parameter set. If a function that we have specialized is invoked more than once with different parameters, then we discard its generated machine code, and fall back into IonMonkey's traditional compilation mode. Even though we might have to recompile a function, our experiments in the SunSpider benchmark suite show that our approach pays for itself. We speed up SunSpider 1.0 by 2.73%. In some cases, as in SunSpider's `access-nsieve.js`, we have been able to achieve a speedup of 38%. We have improved run times in other benchmarks as well: we have observed a 4.8% speedup in V8 version 6, and 1.25% in Kraken 1.1. We emphasize that we are not comparing our prototype against a straw man: our gains have been obtained on top of Mozilla's industrial quality implementation of IonMonkey.

2. A case for value specialization

We propose to specialize the code that the JIT compiler produces for a JavaScript function based on the parameters that this function receives. This kind of optimization is only worth doing if functions are not called many times with different parameters. To check the profitability of this optimization, we have instrumented the Mozilla Firefox browser, and we have used it to collect data from the 100 most visited webpages, according to the Alexa index.[2] We have tried, as much as possible, to use the same methodology as Richards et al. [10]: for each webpage, our script imitates a typical user session, with interactions that simulate keyboard and mouse events. We simulate this mock user section via a jQuery[3] script. This script collects all links and buttons of a webpage and randomly executes them to simulate mouse events. To simulate keyboard interaction we collect all input fields in the webpage, and then fill them with random strings. We have manually navigated through some of these webpages, to certify that our robot produces results that are similar to those that would be obtained by a human being. For each script, we collect information about all function calls invoked during its execution, logging the following information: length in bytecodes, name (including line number of the function definition), and the value plus type of actual arguments.

The histogram in Fig. 1 shows how many times each different JavaScript function is called. This histogram clearly delineates a power distribution. In total we have seen 23,002 different JavaScript functions in the 100 visited websites. Functions with the same name, invoked in different websites, are not considered to be the same; i.e., the internet domain of the website is part of the name of its JavaScript functions. In all, 48.88% of all these functions are called only once during the entire browser session. In all, 11.12% of the functions are called twice. The most invoked functions are from the Kissy UI library, located at Taobao content delivery network (http://a.tbcdn.cn), and the Facebook JavaScipt library (http://static.ak.fbcdn.net). The first one is called 1956 times and the second 1813 times. These numbers indicate that specializing functions to the runtime value of their parameters may be an interesting approach in the JavaScript world.

If we consider functions that are always called with the same parameters, then the distribution is even more concentrated towards 1. The histogram in Fig. 2 shows how often a function is called with different parameters. This experiment shows that 59.91% of all the functions are always called with the same parameters. The descent in this case is impressive, as 8.71% of the functions are called with two different sets of parameters, and 4.60% are called with three. This distribution is more uniform towards the tail than the previous one: the most varied function is called with 1101 different parameters, the second most varied is called with 827, the third most with 736, etc. If it is possible to reuse the same specialized function when its parameters are the same, the histogram in Fig. 2 shows that the speculation that we advocate in this paper succeeds 60% of the time. As we will explain in Section 4, we keep a cache of actual parameter values, so that

[1] This paper is an extended version of earlier work, published in the Symposium on Code Generation and Optimization (CGO) 2013 [9].
[2] Alexa, last visited at http://www.alexa.com, in October 2012.
[3] jQuery Foundation, last visited at `jquery.com/` in October 2012.