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 in 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 speedup SunSpider by 5.38%, and to reduce the size of its native code by 16.72%.

Categories and Subject Descriptors D.3.4 [Processors]: Compilers

General Terms Languages, Performance

Keywords JIT-compilation, JavaScript, Speculation

1. Introduction

JavaScript is presently the most important programming language used in the development of client-side web applications [16]. If in the past only very simple programs would be written in this language, today the reality is different. JavaScript appears in programs ranging from simple form validation routines to applications as complex as Google’s Gmail. Furthermore, JavaScript, in addition to being used directly by developers, is used as the 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 the industry and the academia are putting considerable effort in the creation of efficient execution environments for this programming language.

Nevertheless, executing JavaScript programs efficiently is not an easy task. JavaScript 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 architectures. Thus, compilation time generally has impact on the user experience.

The just-in-time compiler seems to be the tool of choice of engineers to face all these challenges. 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 [9, 15, 21] and type specializers for JavaScript [17]. New optimizations have been proposed to speedup JavaScript programs [27, 31], and old techniques [7] have been put to very good use in state-of-the-art browsers such as Google Chrome. Nevertheless, we believe that the landscape of current just-in-time 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. In Section 3 we revisit a small collection of classic compiler optimizations. 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 function arguments 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 in 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 binaries, and fall back into IonMonkey’s traditional compilation mode. Even though we might have to recompile a function, our experiments show that our approach pays for itself. We speedup SunSpider 1.0 by 5.38%. In some cases, as in SunSpider’s bitops-bitn-in-byte.js, we have been able to achieve a speedup of 49%. We have improved runtimes in other benchmarks as well: we have observed a 4.8% speedup in V8 version 6, and 1.2% in Kraken 1.1. We emphasize that we are comparing our research quality implementation with 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 have used it to collect data from the 100 most visited webpages in the Alexa index. We have tried, as much as possible, to use the same methodology as Richards et al. [14]: for each webpage, our script imitates a typical user section, with interactions that simulate keyboard and mouse events.

The histogram in Figure 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. 48.88% of all these functions are called only once during the entire browser section. 11.12% of the functions are called twice. The two most invoked functions, located at the Taobao content delivery network (http://a.tbcdn.cn), and in the Facebook library (http://static.ak.fbcdn.net), are called 1,956 and 1,813 times. These numbers show that specializing functions to the runtime value of their parameters is a reasonable 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 Figure 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 1,101 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 Figure 2 shows that the speculation that we advocate in this paper is a hit in 60% of the cases. We keep a cache of actual parameter values, so that we can benefit from this regularity. Thus, if the same function is called many times with the same parameters, then we can still run its specialized version.

We have built these histograms to popular benchmarks also. The results are given in Figure 3. The new histograms
We have performed a comparison between the types of the parameters used by functions called with only one set of arguments in the benchmarks, and in the Alexa top 100 websites. The results of this study are shown in Figure 4. Firstly, we observe great diversity between the benchmarks and the web, and among the benchmarks themselves. Nevertheless, one fact is evident: the benchmarks use integers much more often than the JavaScript functions that we found in the wild. 37.5%, 48.72% and 33.03% of the parameters used by functions in SunSpider, V8 and Kraken are integers. On the Internet, only 6.36% of the parameters are integers. In this case, objects and strings are used much more often: 35.57% and 32.95% of the time. Some of the optimizations that we describe in this paper, notably constant propagation, can use integers, doubles and booleans with great benefit: these primitive types allow us to solve some arithmetic operations at code-generation time. We can do less with objects, arrays and strings: we can inline some properties from these types, such as the length constant used in strings. We can also solve some conditional tests, e.g., ==, ===, etc, and we can solve calls to the typeof operator.

3. Parameter Based Value Specialization

Our idea is to replace the parameters of a function by the values that they hold when the function is called. Before explaining how we perform this replacement, we will briefly review the life cycle of a program executed through a just-in-time compiler. In this paper, we focus on the binaries generated by a particular compiler – IonMonkey – yet, the same code layout is used in other JIT engines that combine interpretation with code generation.

Some JavaScript runtime environments, such as Chromium’s V8, compile a function in the first time that function is called. Other runtime systems compile code while this code is interpreted. The Mozilla Firefox engine follows the second approach. A JavaScript function is first interpreted, and then, if a heuristics deems this function worth compiling, it is translated to native code. Figure 5 illustrates this interplay between interpreter and just-in-time compiler. Mozilla’s SpiderMonkey engine comes with a JavaScript interpreter. There exists a number of JIT compilers that work with this interpreter, e.g., TraceMonkey [15], JägerMonkey [17] and IonMonkey. We will be working with the latter.

The journey of a JavaScript function in the Mozilla’s Virtual Machine starts in the Parser, where the JavaScript code is transformed into bytecodes. These bytecodes form a stack-based instruction set, which SpiderMonkey interprets. Some JavaScript functions either are called very often, or contain loops that execute for a long time. We say that these functions are hot. Whenever the execution environment judges a function hot, this function is sent to IonMonkey to be translated to native code.

The JavaScript function, while traversing IonMonkey’s compilation pipeline, is translated into two intermediate representations. The first, the Middle-level Intermediate Representation (MIR) is the baseline format that the compiler optimizes. MIR instructions are three-address code in the static single assignment (SSA) form [12]. This representation provides an infinite supply of virtual registers, also called variables. The main purpose of the MIR format is to provide to the compiler a simple representation that it can optimize. One of the optimizations that IonMonkey does at this level is global value numbering; a task performed via the algorithm.
first described by Alpern et al. [3]. It is at this level that we apply the optimizations that we describe in this section.

The optimized MIR program is converted into another format, before being translated into native code. This format is called the Low-level Intermediate Representation, or LIR. Contrary to MIR, LIR contains machine specific information. MIR’s virtual registers have been replaced by a finite set of names whose purpose is to help the register allocator to find locations to store variables. After IonMonkey is done with register allocation, its code generator produces native binaries. SpiderMonkey diverts its flow to the address of this stream of binary instructions, and the JavaScript engine moves to native execution mode.

This native code will be executed until either it legitimately terminates, or some guard evaluates to false, causing a recompilation. Just-in-time compilers usually speculate on properties of runtime values in order to produce better code. IonMonkey, for instance, uses type specialization. JavaScript represents numbers as infinitely large floating-point values. However, many of these numbers can be represented as simple integers. If the IonMonkey compiler infers that a numeric variable is an integer, then this type is used to compile that variable, instead of the more expensive floating point type. On the other hand, the type of this variable, initially an integer, might change during the execution of the JavaScript program. This modification triggers an event that aborts the execution of the native code, and forces IonMonkey to recompile the entire function.

### 3.1 The Anatomy of a MIR program.

Figure 6 shows an example of a control flow graph (CFG) that IonMonkey produces. In the rest of this paper we will be using a simplified notation that represents the MIR instruction set. Contrary to a traditional CFG, the program in Figure 6 has two entry points. The first, which we have labeled `function entry point` is the path taken whenever the program flow enters the binary function from its beginning. This is the path taken whenever a function already compiled is invoked. The second entry point, the `on stack replacement` (OSR) block, is the path taken by the program flow if the function is translated into binary during its interpretation. As we have mentioned before, a function might be compiled once some heuristics in the interpreter judges that it will run for a long time. In this case, the interpreter must divert the program flow directly to the point that was being interpreted when the native code became active. The OSR block marks this point, usually the first instruction of a basic block that is part of a loop.
The CFG in Figure 6 contains a number of special instructions called `resumepoint`. These instructions indicate places where the state of the program must be saved, so that if it returns to interpretation mode, then the interpreter will not be in an inconsistent state. Resume points are necessary after function calls, for instance, because they might have side effects. On the other hand, referentially transparent commands do not require saving the program state back to the interpreter.

### 3.2 Parameter Specialization

The core optimization that we propose in this paper is parameter specialization. This optimization consists in replacing the arguments passed to a function by the values associated with these arguments at the time the function is called. Our optimizer performs this replacement while the MIR control flow graph is built; therefore, it imposes zero overhead on the compiler. That is, instead of creating a virtual name for each parameter in the graph, we create a constant with that parameter’s runtime value. We have immediate access to the value of each parameter, as it is stored in the interpreter’s stack. There are two types of inputs that we specialize: those in the function entry block, and those in the OSR block. Figure 7(a) shows the effects of this optimization in the program first seen in Figure 6.

### 3.3 Constant Propagation

Constant propagation is, possibly, the most well-known code optimization, and it is described in virtually every compiler textbook. We have implemented the algorithm present in Aho et al.’s classic book [1, p.633-635]. Basically, each program variable is associated with one element in the lattice $\bot < c < \top$, where $c$ is any constant. We iterate successive applications of a meet operator until we reach a fixed point. This meet operator is defined as $\bot \land c = c$, $\bot \land \top = \top$, $\top \land c = \top$, $c_0 \land c_1 = c_0$ if $c_0 = c_1$ and $c_0 \land c_1 = \top$ otherwise.

We have opted for the simplest possible implementation of constant propagation, to reduce the time overhead that our optimization imposes on the runtime environment. Thus, contrary to Wegman et al.’s seminal algorithm [30], we do not extract information from conditional branches.

Figure 7(b) shows the code that results from the application of constant propagation on the program seen in Figure 7(a). If all the arguments of an instruction $i$ are constants, then we can evaluate $i$ at compilation time. If $i$ defines a new variable $v$, then we can replace every use of $v$ by the constant that we have just discovered. The elimination of an instruction that only operates on constants is called folding. We have marked the 14 instructions that we have been able to fold in Figure 7(b). We can fold a large number of JavaScript typical operations. Some of these operations apply only on primitive types, such as numbers, e.g., addition, subtraction, etc. Others, such as the many comparison operators, e.g., $==$, $!=$, $===$, $!==$ and the `typeof` operator, apply on aggregates too.

JavaScript is a very reflective language, and runtime type inspection is a common operation, not only at the development level, but also at the code generation level. As an example, in Figure 6 we check if $s$ is an array, before accessing some of its properties. Our constant propagation allows us to fold away many type guards, which are ubiquitous in the code that IonMonkey generates. We have folded the two type guards in block $L_3$. This optimization is safe, as there is no assignment to variable $s$ in the entire function.

### 3.4 Loop Inversion

Loop inversion is a classic compiler optimization that consists in replacing a while loop by a repeat loop. The main benefit of this transformation is to replace a conditional and an unconditional jump inside a loop by just a conditional loop at its end. Figure 7(c) shows the result of performing loop inversion in the program seen in Figure 7(b). Usually loop inversion inserts a wrapping conditional around the repeat loop, to preserve the semantics of the original program. This conditional, only traversed once by the program flow, ensures that the body of the repeat loop will not be executed if the corresponding while loop iterates zero times.

\[
\text{map}(\text{new Array}(1, 2, 3, 4, 5), 2, 5, \text{inc});
\]

\[
\text{function inc(x)} \{
\quad \text{return } x + 1;
\}\n\]

\[
\text{function map(s, b, n, f)} \{
\quad \text{var } i = b;
\quad \text{while (i < n)} \{
\quad \quad \text{s[i]} = f(s[i]);
\quad \quad i++;
\quad \}\n\quad \text{return } s;
\}\n\]

![Figure 6](image_url)

Figure 6. (a) The JavaScript program that we will use as a running example. (b) The control flow graph of the function map.
Loop inversion does not directly benefit from the knowledge of runtime values. However, we have observed that a subsequent dead-code elimination phase is able to remove the wrapping conditional. This elimination is possible because our parameter specialization often lets us know, at code generation time, that a loop will be executed at least once.

3.5 Dead-Code Elimination

Dead-code elimination removes instructions that we prove that cannot be reached by the program flow. We run it after constant propagation, in order to give instruction folding the chance to transform conditional branches into simple boolean values. Whenever this extensive folding is possible, the outcome of the conditional branch can be predicted at compile-time; thus, we can safely remove the branch instruction and, possibly, blocks of unreachable code. Figure 8(a) shows the effects of dead-code elimination on the program in Figure 7(c). We have removed block L2, because the result of the comparison inside this block is known at code generation time. Notice that we keep the function entry block. We only keep this block because we can cache the binaries that we produce, in case a function is called with the same parameters again. If compiled function is called again, then execution must start at that function’s entry point.

3.6 Array Bounds Check Elimination

JavaScript is a type safe language, which means that any value can only be used according to the contract specified by its runtime type. As a consequence of this type safety, array accesses in JavaScript are bound checked. Accesses outside the bounds of the array return the `undefined` constant, which is the only element in the Undefined data type. Bound checking an index i is a relatively expensive operation, because, at the native code level it requires loading the array length property l, and demands two conditional tests: i ≥ 0 and i < l. The knowledge of function inputs allows us to eliminate some simple bound checks.

To perform this optimization, we need to identify integer variables that control loops. If these induction variables are bounded by a known value, then we can perform a trivial kind of range analysis to estimate the minimum and maximum values that array indices might receive. In order to keep our optimizer simple and efficient, we only recognize variables defined by the pattern $i_0 = \text{exp}; i_1 = \phi(i_0, i_2); i_2 = i_1 + c_2$. Variables $i_3$ and $i_4$, plus the constant 2, in Figure 8(a) follow this pattern. Variable $i_2$ is initially assigned the constant 2, and $i_2 < 100$ inside the loop; hence, its range is [2, 99]. Moreover, $i_4 = \phi(2, i_2)$; thus, its range is also [2, 99]. Therefore, any access of array $s_2$, e.g., reference 0xFF3D8800 in the figure, indexed by $i_4$ is safe, as $s_2$’s length is 100. Figure 8(b) shows the result of eliminating the bounds checks from the program in Figure 8(b).

3.7 Function Inlining

JavaScript supports closures; therefore, it is possible to pass a function as an argument to another one. We inline functions passed as arguments, whenever possible. We are also able to inline methods from objects that are passed as parameters. Figure 8(c) shows the result of replacing the call to function `inc`, seen in Figure 8(b), by its body. IonMonkey
4. Experiments

The experiments that we describe in this section were performed on a Quad-Core Intel i5 processor with 3.3GHz of clock and 8GB of RAM running Ubuntu 12.04.1 32-bits. The IonMonkey version that we are using was obtained from the Mozilla repository on August 3rd, 2012.

The benchmarks. Timing JavaScript applications in actual webpages is not trivial, because too many factors, mostly related to I/O operations, have an impact on the runtime of these programs. Therefore, we chose to use three well know benchmarks: SunSpider, Kraken and V8 in our experiments. The advantage of the benchmarks is that we can measure their execution time reliably. The disadvantage is that, as pointed by Richards et al. [14], benchmarks might not reflect the true nature of the JavaScript applications found in the wild. The data that we shown in Section 2 is a testimony of this fact. On one hand, Figure 4 shows that actual applications tend to use instances of object more often than instances of simpler types. The optimizations in Section 3 work better in the latter case. On the other hand, Figures 1, 2 and 3 seem to imply that functions tend to be called with the same parameters more often in the wild, then in benchmarks.

The impact of parameter specialization on runtime. Figure 9 (a-b) shows the impact of our specialization in the runtime of three different benchmark suites. We have executed each of our benchmarks 100 times, to reduce the imprecision of this experiment. The time measured in each run includes interpretation, compilation and native execution. We have tested different configurations of the optimizations that we have implemented. PARAMETERSPEC is the parameter specialization that we have described in Section 3.2, augmented with the automatic inlining of functions passed as parameters, as we explained in Section 3.7.

We have run constant propagation without parameter specialization, as we show in the third column of the table, observing a runtime slowdown of (-1.04%). Without parameter specialization, constant propagation has little room to improve the code, as IonMonkey’s global value numbering already eliminates most of the constants in the scripts. On the other hand, the combination of parameter specialization, constant propagation and dead-code elimination has produced one of our best results. Some optimizations enable others. As an example, in SunSpider’s string-unpack-code, loop inversion has improved the effectiveness of IonMonkey’s invariant code motion, yielding a 28% speedup. Our implementation of array bounds check elimination did not give us a substantial speedup in any benchmark. We are using a simple approach, i.e., we only eliminate checks from arrays indexed by induction variables. Moreover, the alias analysis that ensures the correctness of this optimization is currently implemented in IonMonkey in a very simple way. Thus, if there exists any store instruction in the script being compiled, the elimination of bound check instructions is considered unsafe and is not performed. If
<table>
<thead>
<tr>
<th>Parameter Spec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Prop</td>
</tr>
<tr>
<td>Loop Inversion</td>
</tr>
<tr>
<td>Dead Code Elim</td>
</tr>
<tr>
<td>Bound Check Elim</td>
</tr>
</tbody>
</table>

- (a) Overall runtime speedup (% arithmetic mean)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SunSpider 1.0</th>
<th>V8 Version 6</th>
<th>Kraken 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup (%)</td>
<td>4.81</td>
<td>4.00</td>
<td>0.75</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>-1.04</td>
<td>-0.50</td>
<td>-0.08</td>
</tr>
<tr>
<td>Code Size (%)</td>
<td>4.46</td>
<td>4.17</td>
<td>0.75</td>
</tr>
<tr>
<td>Code Size (%)</td>
<td>4.62</td>
<td>4.33</td>
<td>0.83</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>5.35</td>
<td>2.00</td>
<td>1.25</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>5.12</td>
<td>4.83</td>
<td>0.75</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.12</td>
<td>4.00</td>
<td>0.67</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>5.12</td>
<td>1.17</td>
<td>0.67</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>5.38</td>
<td>4.67</td>
<td>0.50</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.54</td>
<td>2.50</td>
<td>0.75</td>
</tr>
</tbody>
</table>

- (b) Overall runtime speedup (% geometric mean)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SunSpider 1.0</th>
<th>V8 Version 6</th>
<th>Kraken 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speedup (%)</td>
<td>4.45</td>
<td>3.92</td>
<td>0.74</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>-1.04</td>
<td>-0.50</td>
<td>-0.08</td>
</tr>
<tr>
<td>Code Size (%)</td>
<td>4.03</td>
<td>4.09</td>
<td>0.74</td>
</tr>
<tr>
<td>Code Size (%)</td>
<td>4.21</td>
<td>4.25</td>
<td>0.83</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.91</td>
<td>1.74</td>
<td>1.24</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.51</td>
<td>4.75</td>
<td>0.74</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>3.68</td>
<td>3.92</td>
<td>0.66</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.59</td>
<td>0.94</td>
<td>0.66</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.80</td>
<td>4.59</td>
<td>0.49</td>
</tr>
<tr>
<td>Speedup (%)</td>
<td>4.13</td>
<td>2.31</td>
<td>0.74</td>
</tr>
</tbody>
</table>

- (c) Compilation overhead (% arithmetic mean)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SunSpider 1.0</th>
<th>V8 Version 6</th>
<th>Kraken 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>-7.2</td>
<td>1.5</td>
<td>-3.0</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>3.6</td>
<td>2.1</td>
<td>3.0</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-4.3</td>
<td>4.0</td>
<td>-0.9</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-6.5</td>
<td>2.6</td>
<td>-2.4</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-7.3</td>
<td>1.7</td>
<td>-0.7</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-4.0</td>
<td>3.6</td>
<td>16.2</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-3.8</td>
<td>4.3</td>
<td>-0.5</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-4.5</td>
<td>3.4</td>
<td>16.5</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-3.6</td>
<td>3.8</td>
<td>1.4</td>
</tr>
</tbody>
</table>

- (d) Compilation overhead (% geometric mean)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>SunSpider 1.0</th>
<th>V8 Version 6</th>
<th>Kraken 1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%)</td>
<td>-8.7</td>
<td>1.4</td>
<td>-5.1</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>3.6</td>
<td>2.1</td>
<td>3.0</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-5.8</td>
<td>3.9</td>
<td>-2.9</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-8.0</td>
<td>2.4</td>
<td>-4.4</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-8.7</td>
<td>1.6</td>
<td>-2.9</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-5.5</td>
<td>3.5</td>
<td>5.2</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-5.4</td>
<td>4.2</td>
<td>-2.5</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-6.0</td>
<td>3.2</td>
<td>-0.5</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-5.1</td>
<td>3.7</td>
<td>5.6</td>
</tr>
<tr>
<td>Overhead (%)</td>
<td>-5.4</td>
<td>3.2</td>
<td>0.5</td>
</tr>
</tbody>
</table>

Figure 9. (a-b) Percentage of runtime speedup for each test of our benchmark suites, considering: parameter specialization (Section 3.2), constant propagation (Section 3.3), loop inversion (Section 3.4), dead code elimination (Section 3.5) and array bounds check elimination (Section 3.6). (c-d) Percentage of compilation overhead, considering the same set of optimizations.

we run all our optimizations together, we do not obtain the best speedups. This happens because these optimizations are not cumulative: dead code elimination and bound check elimination may, for instance, eliminate the same code.

Size of Generated Code. One of the benefits of our optimizations is code size reduction, which results from the combination of parameter specialization with dead code elimination. Figure 10 presents the size of the machine code generated for functions of our three benchmarks with and without our approach. Notice that, due to recompi-
lations, the same function may be translated in different ways. In this analysis, we consider only the smallest version that each compilation mode generates for each function. On average, we are able to reduce the size of the functions in SunSpider by 16.72%. This reduction, for V8 and Kraken, is 18.84% and 15.94%. These numbers translate to real-world applications. We have run our techniques on the JavaScript benchmark automatically built from actual web-
pages, using Richards et al.’s tool [23], obtaining a code-
size reduction of 12.07% for www.google.com, 16.08% for www.facebook.com, and 22.10% for www.twitter.com. In the first benchmark, we recompile 5.0% more functions, in the second, 4.9%, and in the last one, 23.1%.

Compilation overhead. Figure 9 (c-d) shows the impact of our optimizations in IonMonkey’s compilation time, i.e., the
time that this engine spends analyzing, optimizing and generating code. Surprisingly, many of our configurations improve IonMonkey’s compilation time. As we have seen in Figure 10, our optimizations decrease function sizes; thus, reducing the amount of work performed by the other phases of IonMonkey. Our key optimization, parameter specialization, has no overhead by construction. Instead of generating instructions that manipulate memory locations, we do it for constants. Additionally, this optimization improves the time of the register allocator, given that it reduces register pres-
sure substantially.

Specialization policy. We specialize every function compiled by IonMonkey, i.e., hot functions, and we cache the arguments of each specialized function. In this way, if a function is called in sequence with the same arguments, then we reuse the specialized code. We distinguish successfully specialized and deoptimized functions. The former category represents the functions that are always called, throughout the entire program execution, with the same arguments. In this case, we have a win-win condition: we produce more efficient code, and do not have to discard it later. If a function is called again with different arguments, then we discard its specialized code, and recompile it, this time producing generic code for it. Additionally, the function is marked and we do not try to specialize it again. We have special-
the strategies that JIT compilers use to specialize code at runtime. For a comprehensive survey on just-in-time compilation, we recommend the work of John Aycock [5].

Control Flow Speculation. There are two types of speculation usually seen in just-in-time compilers: data and control speculation. Data speculation happens whenever the compiler assumes that a certain data has a particular property. This paper describes a data speculation technique. Control speculation happens if the compiler assumes that a path will be taken, and generates better code for that path. An example of control speculation has been recently proposed by Mehrara and Mahlke [21] in the context of trace compilers. The authors propose to execute the main flow of a program’s trace speculatively, delegating to a second thread the task of checking if any side-exit has been taken. Whenever such event is detected, the two threads synchronize, the state of the interpreter is recovered, and execution falls back into interpretation mode.

Input-Oriented Compilation. Tian et al. describe a general code generation paradigm that they call Input-Centric [28, 29]. In this model, the compiler generates the best possible code for a certain universe of possible inputs. Machine learning might be used to determine which strategy the compiler should adopt to produce code. A recent example of input-oriented compilation is the work of Samadi et al. [26]. The authors generate programs containing distinct sub-programs to handle different kinds of inputs. Runtime characteristics of the particular input determine which routine will be activated. The input aware compiler has the advantage of being more general than our approach: it generates code that works on different inputs. Our code only works with certain values. However, the optimizations that we can perform are more extensive: we trade generality for over-specialization.

5. Related Work

Just-in-time compilers are part of the programming language’s folklore since the early 60’s. Influenced by the towering work of John McCarthy [20], the father of Lisp, a multitude of JIT compilers have been designed and implemented. These compilers have been fundamental to the success of languages such as Smalltalk [13], Self [7], Python [24], Java [18], and many others. In this section we will focus on the strategies that JIT compilers use to specialize code at runtime.
Nöel [11], for instance, propose to fill the templates with data extracted from the input values passed to the program. The static analysis that Consel and Nöel use to build the templates classifies variables as either static (known) or dynamic (unknown). The variables in the former group define the overall structure of the template, whereas the variables in the latter determine the parts of this structure that must be completed dynamically. The main difference between template-based compilation and the approach that we advocate in this paper is that we do not require the pre-compilation phase in which templates are built. Moreover, as a technical detail, we specialize code at the function level, i.e., we use the values passed to individual functions, whereas these related works did not try to implement any other optimization besides parameter specialization. Our preliminary implementation was based on the work of Sol et al. [27]. Sol et al. have proposed an overflow-check elimination technique for JIT compilers. JavaScript represents numbers as floating point values. Gal et al. [15], as mentioned before, replace these numbers by integers whenever possible. However, overflow guards are inserted along the binary code, to preserve the correct semantics of JavaScript. Sol et al. have designed a range analysis to eliminate some of these guards, and have implemented it in Mozilla’s TraceMonkey. This range analysis inspects the values piled on the interpreter’s stack for improved precision. Sol et al. have shown that this knowledge adds an almost three-fold increase in the effectiveness of the overflow elimination routine, when compared to a completely static approach, such as Rodrigues et al.’s [25]. The scope of our paper is broader than Sol’s: we specialize whole functions, instead of traces, and use several compiler optimizations, instead of only overflow-check elimination.

There is a large body of work related to runtime value specialization. The core idea is simple: once the compiler proves that a value belongs into a certain type, it uses this type directly, instead of resorting to costly boxing and unboxing operations. Due to the speculative nature of these optimizations, guards are inserted throughout the binary code, so that, in case the type of a variable changes, the execution environment can adapt accordingly. Armin Rigo provides an extensive description about this form of specialization, in the context of Pyco, a JIT compiler for Python [24]. Almasi et al. [2] describe a complete compilation framework for Matlab that does type specialization. They assume that the type of the input values passed to a function will not change; hence, they generate binaries customized to these types. Also in the Matlab world, yet more recently, Boisvert et al. [10] have used type specialization to better handle arrays and matrices.

In another front, the researchers from the Mozilla Foundation have proposed several ways to apply type information onto JavaScript code [8, 15, 17]. Gal et al. [15]’s TraceMonkey performs a number of type-based optimizations, such as converting floating point numbers to integers, and sparse objects to dense vectors. Type specialization seems to be, in fact, one of the key players in the Firefox side of the browser war, as Hackett and Shu have recently demonstrated [17].

Runtime Value Specialization The work of Bolz et al. [6] is close to ours. The authors propose to use just-in-time partial evaluation to speed up Prolog programs. Conceptually, Bolz’s approach differs from ours in two main ways. First, we transform programs by using classic compiler optimizations, whereas Bolz et al. use standard partial evaluation. Second, similar to the template-based approach, Bolz et al. rely on a pre-compilation phase, in which they create hooks that the partial evaluator uses to manipulate the program. Thus, whereas we do not require a pre-compilation phase, Bolz et al. preprocess the program before running it.

The present work completes our first attempt to implement value specialization in IonMonkey [4]. That endeavor was a very limited study: our compiler would abort execution if the same function was called more than once, and we did not try to implement any other optimization besides parameter specialization. Our preliminary implementation was based on the work of Sol et al. [27]. Sol et al. have proposed an overflow-check elimination technique for JIT compilers. JavaScript represents numbers as floating point values. Gal et al. [15], as mentioned before, replace these numbers by integers whenever possible. However, overflow guards are inserted along the binary code, to preserve the correct semantics of JavaScript. Sol et al. have designed a range analysis to eliminate some of these guards, and have implemented it in Mozilla’s TraceMonkey. This range analysis inspects the values piled on the interpreter’s stack for improved precision. Sol et al. have shown that this knowledge adds an almost three-fold increase in the effectiveness of the overflow elimination routine, when compared to a completely static approach, such as Rodrigues et al.’s [25]. The scope of our paper is broader than Sol’s: we specialize whole functions, instead of traces, and use several compiler optimizations, instead of only overflow-check elimination.

6. Conclusion

This paper has described a suite of value-based compiler optimizations that we have used to improve the execution time of an industrial-strength just-in-time compiler. We believe that this paper has pushed further the notion of JIT speculation, as we produce highly-specialized binaries given the input values passed to a function. Our approach is most profitable when applied on functions that are called always with the same parameters. We have demonstrated that these calls are very frequent, be it in well-known benchmarks, be it in actual webpages. We hope that the code optimization approach that we advocate in this paper will open new directions to compiler research. It is our intention to re-implement other classic compiler optimizations such as loop-unrolling and overflow-check elimination in the context of runtime-value specialization. We also plan to experiment our approach with different heuristics. For instance, we cache only one binary per function. Thus, we can specialize only two different parameter sets for the same function. We believe that this approach is the best tradeoff, given the behavior of the JavaScript programs that we have found; however, more experiments are necessary to confirm this hypothesis.

Software: our code, plus all the data that we have used in this paper is available at our repository.

Acknowledgments

We thank Nicolas Pierton, David Anderson and all the Mozilla engineers for their support, without which this work would not be possible. We also thank the anonymous reviewers for their useful comments and suggestions, and our shepherd, Robert Hundt. This project is supported by FAPEMIG and InWeb.

http://code.google.com/p/jit-value-specialization
References


