# Just-in-Time Value Specialization

Igor Costa, Péricles Alves, Henrique Nazaré Santos, Fernando Magno Quintão Pereira

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

{igor,periclesrafael,hnsantos,fernando}@dcc.ufmg.br

## 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

*CGO '13* 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

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-bits-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 [1]. 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.

[1] `http://www.alexa.com`

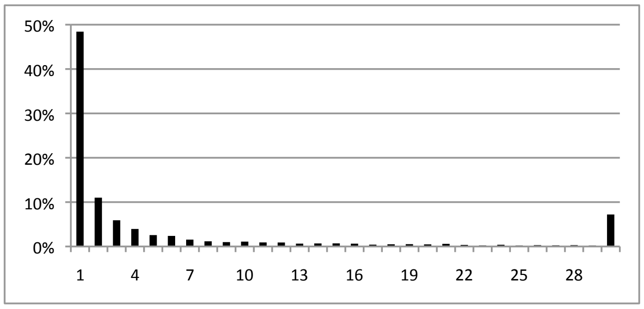


**Figure 1.** Histogram showing the percentage of JavaScript functions (Y-axis) that are called $n$ times (X-axis). Data taken from Alexa's 100 top websites. The histogram has 353 entries; however, we only show the first 29. The tail has been combined in entry 30.

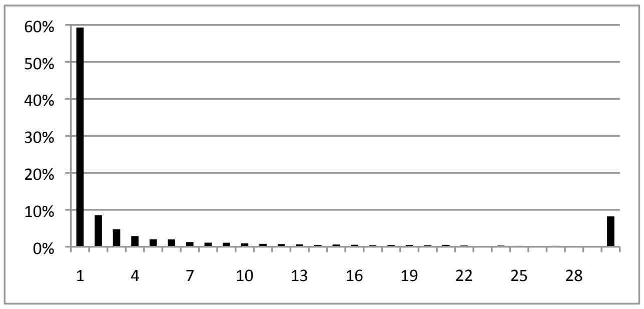


**Figure 2.** Histogram showing the percentage of JavaScript functions (Y-axis) that are called with $n$ different sets of arguments (X-axis). Data taken from Alexa's 100 top websites. We only show the first 29 entries.

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

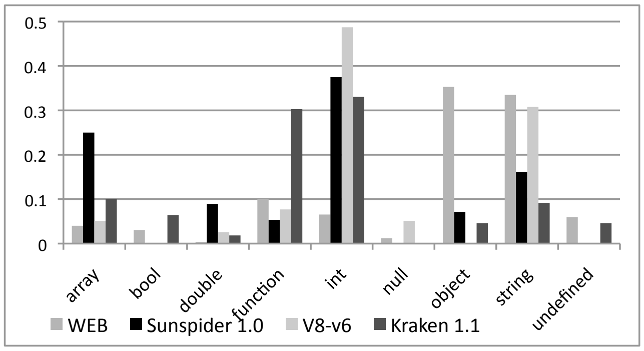


**Figure 4.** The most common types of parameters used in benchmarks and in actual webpages.

are more varied than in the previous analysis. We speculate that this greater diversity happens because we are considering a universe with much less elements: We have 154 distinct functions in SunSpider, 186 in Kraken, and 320 in Google's V8. Nevertheless, we can still observe a power law, mainly in SunSpider's and Kraken's distribution. 21.43% of SunSpider's functions are called only once. This number is only 4.68% in V8, but is 39.79% in Kraken. The function most often called in SunSpider, `md5_ii` from the crypto-md5 benchmark, was invoked 2,300 times. In V8 we have observed 3,209 calls of the method `sc_Pair` in the `earley-boyer` benchmark. In Kraken, the most called function is in `stanford-crypto-ccm`, an anonymous function invoked 648 times.

If we consider how often each function is invoked with the same parameters, then we have a more evident power distribution. Figure 3 (Bottom) shows these histograms. We have that 38.96% of the functions are called with the same actual parameters in SunSpider, 40.62% in V8, and 55.91% in Kraken. At least for V8 we have a stark contrast with the number of invocations of the same function: only 4.68% of the functions are called a single time, yet the number of functions invoked with the same arguments is one order of magnitude larger. In the three collections of benchmarks, the most called functions are also the most varied ones. In SunSpider, each of the 2,300 calls of the `md5_ii` function receives different values. In V8 and Kraken, the most invoked functions were called with 2,641 and 643 different parameter sets, respectively.

**The types of the parameters:** 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

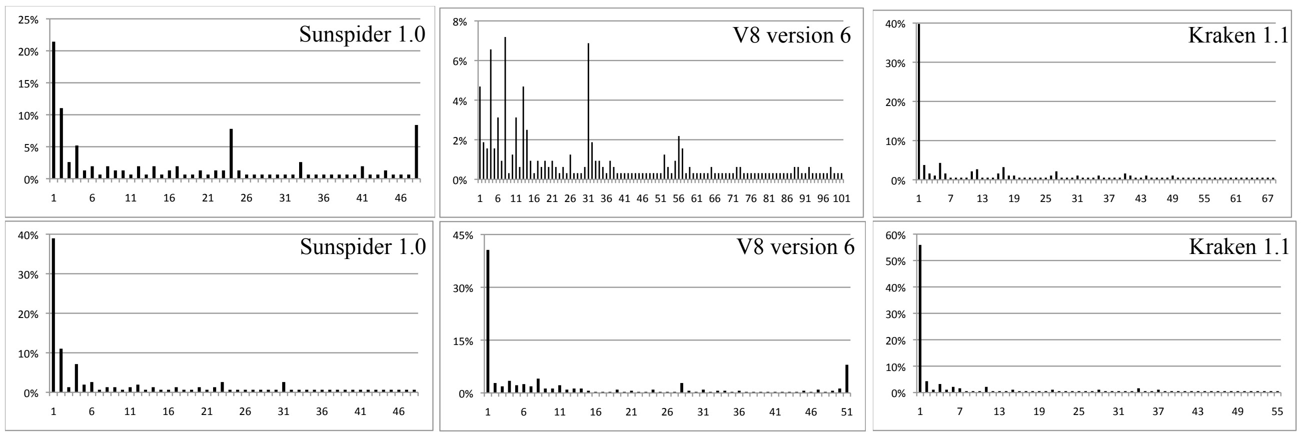
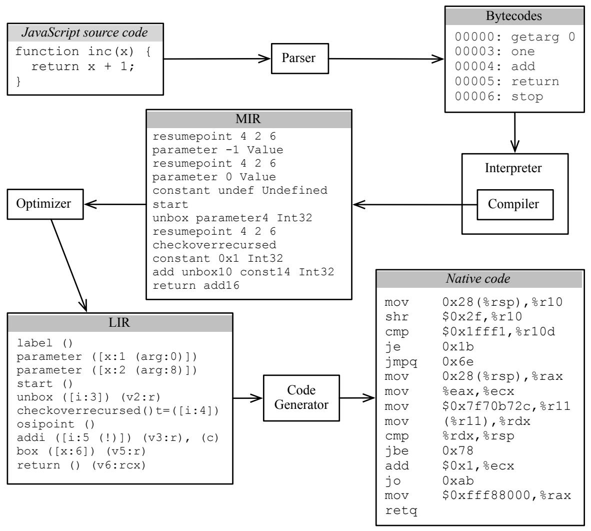


**Figure 3.** Invocation histograms for three different benchmark suites. (Top) Fraction of the number of times that each function is called. (Bottom) Fraction of the number of times that each function is called with the same parameters.



*JavaScript source code*
```
function inc(x) {
  return x + 1;
}
```

Bytecodes
```
00000: getarg 0
00003: one
00004: add
00005: return
00006: stop
```

MIR
```
resumepoint 4 2 6
parameter -1 Value
resumepoint 4 2 6
parameter 0 Value
constant undef Undefined
start
unbox parameter4 Int32
resumepoint 4 2 6
checkoverrecursed
constant 0x1 Int32
add unbox10 const14 Int32
return add16
```

LIR
```
label ()
parameter ([x:1 (arg:0)])
parameter ([x:2 (arg:8)])
start ()
unbox ([i:3]) (v2:r)
checkoverrecursed()t=([i:4])
osipoint ()
addi ([i:5 (!)]) (v3:r), (c)
box ([x:6]) (v5:r)
return () (v6:rcx)
```

*Native code*
```
mov     0x28(%rsp),%r10
shr     $0x2f,%r10
cmp     $0x1fff1,%r10d
je      0x1b
jmpq    0x6e
mov     0x28(%rsp),%rax
mov     %eax,%ecx
mov     $0x7f70b72c,%r11
mov     (%r11),%rdx
cmp     %rdx,%rsp
jbe     0x78
add     $0x1,%ecx
jo      0xab
mov     $0xfff88000,%rax
retq
```

**Figure 5.** The life cycle of a JavaScript program in the SpiderMonkey/IonMonkey execution environment.

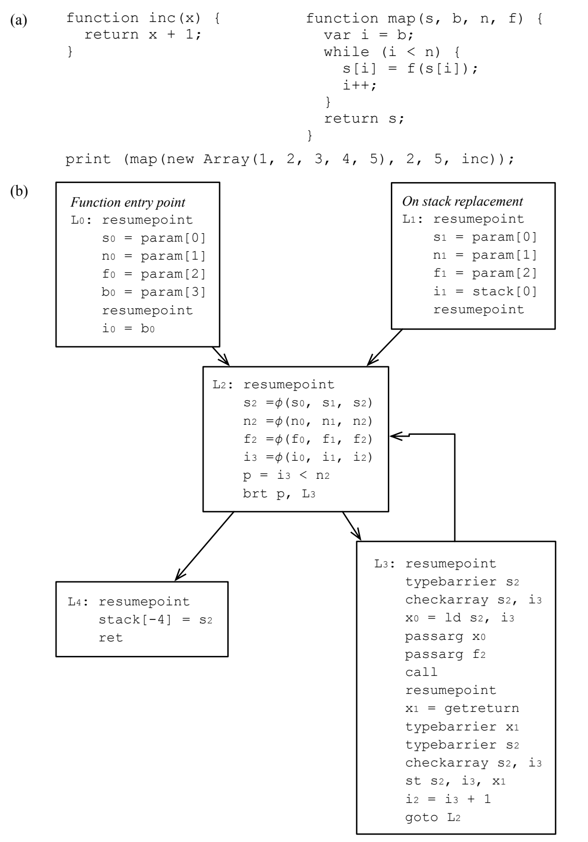
first described by Alpern *et al.* [3]. It is at this level that we apply the optimizations that we describe in this section.

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.



(a)

```
function inc(x) {          function map(s, b, n, f) {
  return x + 1;              var i = b;
}                            while (i < n) {
                               s[i] = f(s[i]);
                               i++;
                             }
                             return s;
                           }
print (map(new Array(1, 2, 3, 4, 5), 2, 5, inc));
```

(b)

```
Function entry point
L0: resumepoint
    s0 = param[0]
    n0 = param[1]
    f0 = param[2]
    b0 = param[3]
    resumepoint
    i0 = b0
```

```
On stack replacement
L1: resumepoint
    s1 = param[0]
    n1 = param[1]
    f1 = param[2]
    i1 = stack[0]
    resumepoint
```

```
L2: resumepoint
    s2 =ϕ(s0, s1, s2)
    n2 =ϕ(n0, n1, n2)
    f2 =ϕ(f0, f1, f2)
    i3 =ϕ(i0, i1, i2)
    p = i3 < n2
    brt p, L3
```

```
L4: resumepoint
    stack[-4] = s2
    ret
```

```
L3: resumepoint
    typebarrier s2
    checkarray s2, i3
    x0 = ld s2, i3
    passarg x0
    passarg f2
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    typebarrier s2
    checkarray s2, i3
    st s2, i3, x1
    i2 = i3 + 1
    goto L2
```

**Figure 6.** (a) The JavaScript program that we will use as a running example. (b) The control flow graph of the function `map`.

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

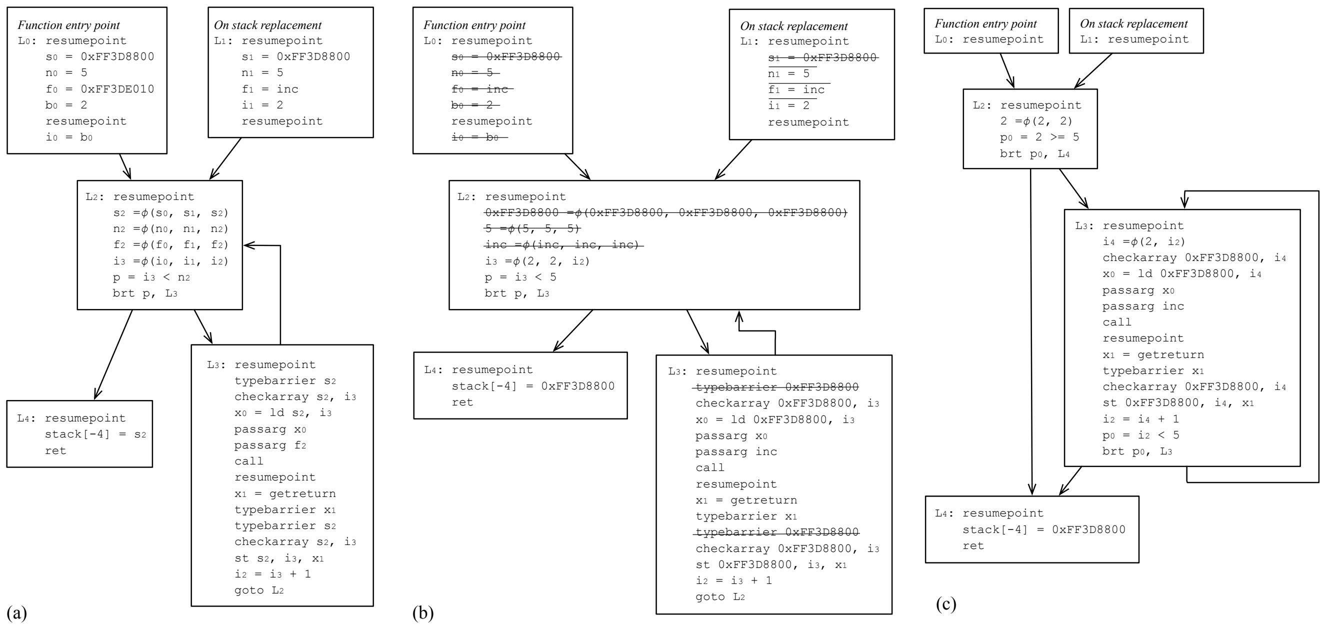
Constant propagation is, possibly, the most well-known code optimization, and it is described in virtually every compiler textbook. We have implemented the algorithm present in Aho *et al.*'s classic book [1, p.633-635]. Basically, each program variable is associated with one element in the lattice $\bot < c < \top$, 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 $\bot \wedge c = c$, $\bot \wedge \top = \top$, $\top \wedge c = \top$, $c_0 \wedge c_1 = c_0$ if $c_0 = c_1$ and $c_0 \wedge c_1 = \top$ 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.



```
(a)
Function entry point
L0: resumepoint
    s0 = 0xFF3D8800
    n0 = 5
    f0 = 0xFF3DE010
    b0 = 2
    resumepoint
    i0 = b0

On stack replacement
L1: resumepoint
    s1 = 0xFF3D8800
    n1 = 5
    f1 = inc
    i1 = 2
    resumepoint

L2: resumepoint
    s2 =φ(s0, s1, s2)
    n2 =φ(n0, n1, n2)
    f2 =φ(f0, f1, f2)
    i3 =φ(i0, i1, i2)
    p = i3 < n2
    brt p, L3

L4: resumepoint
    stack[-4] = s2
    ret

L3: resumepoint
    typebarrier s2
    checkarray s2, i3
    x0 = ld s2, i3
    passarg x0
    passarg f2
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    typebarrier s2
    checkarray s2, i3
    st s2, i3, x1
    i2 = i3 + 1
    goto L2
```

```
(b)
Function entry point
L0: resumepoint
    ~~s0 = 0xFF3D8800~~
    ~~n0 = 5~~
    ~~f0 = inc~~
    ~~b0 = 2~~
    resumepoint
    ~~i0 = b0~~

On stack replacement
L1: resumepoint
    ~~s1 = 0xFF3D8800~~
    n1 = 5
    f1 = inc
    i1 = 2
    resumepoint

L2: resumepoint
    ~~0xFF3D8800 =φ(0xFF3D8800, 0xFF3D8800, 0xFF3D8800)~~
    ~~5 =φ(5, 5, 5)~~
    ~~inc =φ(inc, inc, inc)~~
    i3 =φ(2, 2, i2)
    p = i3 < 5
    brt p, L3

L4: resumepoint
    stack[-4] = 0xFF3D8800
    ret

L3: resumepoint
    ~~typebarrier 0xFF3D8800~~
    checkarray 0xFF3D8800, i3
    x0 = ld 0xFF3D8800, i3
    passarg x0
    passarg inc
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    ~~typebarrier 0xFF3D8800~~
    checkarray 0xFF3D8800, i3
    st 0xFF3D8800, i3, x1
    i2 = i3 + 1
    goto L2
```

```
(c)
Function entry point
L0: resumepoint

On stack replacement
L1: resumepoint

L2: resumepoint
    2 =φ(2, 2)
    p0 = 2 >= 5
    brt p0, L4

L3: resumepoint
    i4 =φ(2, i2)
    checkarray 0xFF3D8800, i4
    x0 = ld 0xFF3D8800, i4
    passarg x0
    passarg inc
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    checkarray 0xFF3D8800, i4
    st 0xFF3D8800, i4, x1
    i2 = i4 + 1
    p0 = i2 < 5
    brt p0, L3

L4: resumepoint
    stack[-4] = 0xFF3D8800
    ret
```

**Figure 7.** (a) The result of our parameter specialization applied onto the program in Figure 6. (b) Constant propagation. (c) Loop inversion

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 $L_2$, 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.

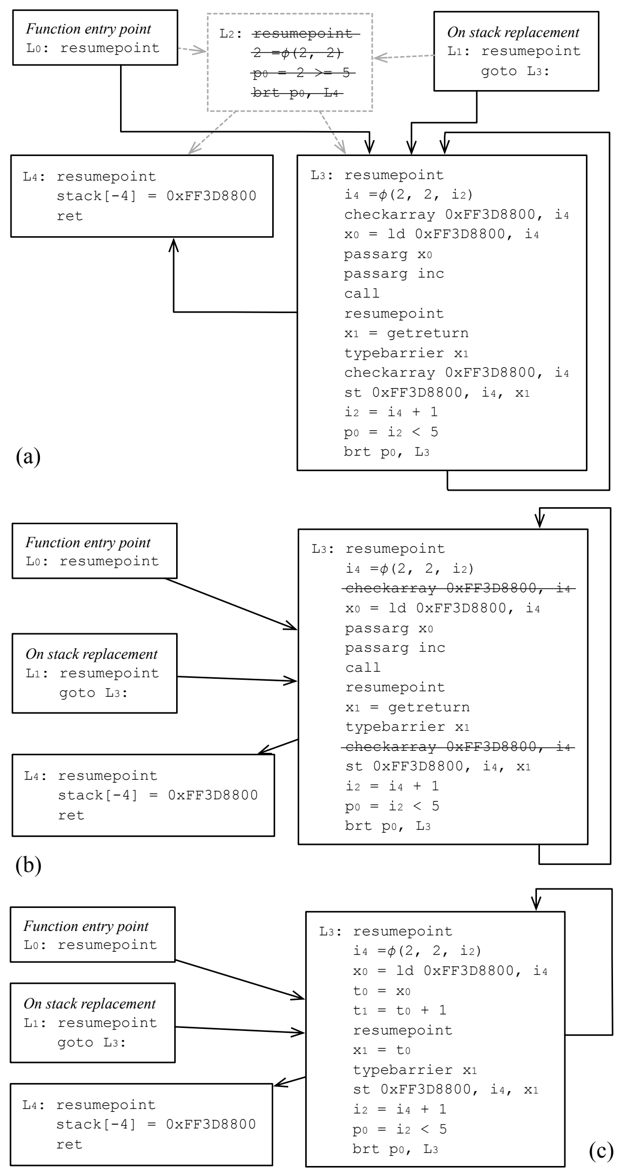
## 3.6 Array Bounds Check Elimination

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: $i \geq 0$ 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 $i_0 = exp; i_1 = \phi(i_0, i_2); i_2 = i_1 + c_2$. 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



```
Function entry point
L0: resumepoint
```

```
L2: ~~resumepoint~~
    ~~2 =ϕ(2, 2)~~
    ~~p0 = 2 >= 5~~
    ~~brt p0, L4~~
```

```
On stack replacement
L1: resumepoint
    goto L3:
```

```
L4: resumepoint
    stack[-4] = 0xFF3D8800
    ret
```

```
L3: resumepoint
    i4 =ϕ(2, 2, i2)
    checkarray 0xFF3D8800, i4
    x0 = ld 0xFF3D8800, i4
    passarg x0
    passarg inc
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    checkarray 0xFF3D8800, i4
    st 0xFF3D8800, i4, x1
    i2 = i4 + 1
    p0 = i2 < 5
    brt p0, L3
```

```
Function entry point
L0: resumepoint
```

```
On stack replacement
L1: resumepoint
    goto L3:
```

```
L4: resumepoint
    stack[-4] = 0xFF3D8800
    ret
```

```
L3: resumepoint
    i4 =ϕ(2, 2, i2)
    ~~checkarray 0xFF3D8800, i4~~
    x0 = ld 0xFF3D8800, i4
    passarg x0
    passarg inc
    call
    resumepoint
    x1 = getreturn
    typebarrier x1
    ~~checkarray 0xFF3D8800, i4~~
    st 0xFF3D8800, i4, x1
    i2 = i4 + 1
    p0 = i2 < 5
    brt p0, L3
```

```
Function entry point
L0: resumepoint
```

```
On stack replacement
L1: resumepoint
    goto L3:
```

```
L4: resumepoint
    stack[-4] = 0xFF3D8800
    ret
```

```
L3: resumepoint
    i4 =ϕ(2, 2, i2)
    x0 = ld 0xFF3D8800, i4
    t0 = x0
    t1 = t0 + 1
    resumepoint
    x1 = t0
    typebarrier x1
    st 0xFF3D8800, i4, x1
    i2 = i4 + 1
    p0 = i2 < 5
    brt p0, L3
```

**Figure 8.** (a) Dead code elimination. (b) Array bounds check elimination. (c) Function inlining.

already performs function inlining; however, it does it after 40,000 calls, much later than we do. IonMonkey's inliner is profile guided: after a function is called a large number of times, it decides to inline it. Closures are not immediately inlined, as they are passed as formal parameters to a function that can be called with many different actual parameters. Furthermore, inlining closures requires guards: if the host function is called again, this time with a different closure, recompilation must take place. Our aggressive approach to inlining avoids all this burden. We inline a closure as soon as we compile the host function, and we do not use guards. In case the function is called again, our entire code will be discarded; hence, these guards would not be necessary.

## 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 gives 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

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| PARAMETERSPEC | ● | | ● | ● | ● | ● | ● | ● | ● | ● |
| CONSTANTPROPG | | ● | ● | | | ● | ● | ● | ● | ● |
| LOOPINVERSION | | | | | ● | | | ● | | ● |
| DEADCODEELIM | | | | ● | | ● | | | ● | ● |
| BOUNDCHECKELIM | | | | | | | ● | | ● | ● |
| – (a) Overall runtime speedup (% arithmetic mean) – | | | | | | | | | | |
| SunSpider 1.0 | 4.81 | -1.04 | 4.46 | 4.62 | 5.35 | 5.12 | 4.12 | 5.12 | 5.38 | 4.54 |
| V8 version 6 | 4.00 | -0.50 | 4.17 | 4.33 | 2.00 | 4.83 | 4.00 | 1.17 | 4.67 | 2.50 |
| Kraken 1.1 | 0.75 | -0.08 | 0.75 | 0.83 | 1.25 | 0.75 | 0.67 | 0.67 | 0.50 | 0.75 |
| – (b) Overall runtime speedup (% geometric mean) – | | | | | | | | | | |
| SunSpider 1.0 | 4.45 | -1.04 | 4.03 | 4.21 | 4.91 | 4.51 | 3.68 | 4.59 | 4.80 | 4.13 |
| V8 version 6 | 3.92 | -0.50 | 4.09 | 4.25 | 1.74 | 4.75 | 3.92 | 0.94 | 4.59 | 2.31 |
| Kraken 1.1 | 0.74 | -0.08 | 0.74 | 0.83 | 1.24 | 0.74 | 0.66 | 0.66 | 0.49 | 0.74 |
| – (c) Compilation overhead (% arithmetic mean) – | | | | | | | | | | |
| Sunspider 1.0 | -7.2 | 3.6 | -4.3 | -6.5 | -7.3 | -4.0 | -3.8 | -4.5 | -3.6 | -3.9 |
| V8 version 6 | 1.5 | 2.1 | 4.0 | 2.6 | 1.7 | 3.6 | 4.3 | 3.4 | 3.8 | 3.4 |
| Kraken 1.1 | -3.0 | 3.0 | -0.9 | -2.4 | -0.7 | 16.2 | -0.5 | 1.6 | 16.5 | 1.4 |
| – (d) Compilation overhead (% geometric mean) – | | | | | | | | | | |
| Sunspider 1.0 | -8.7 | 3.6 | -5.8 | -8.0 | -8.7 | -5.5 | -5.4 | -6.0 | -5.1 | -5.4 |
| V8 version v6 | 1.4 | 2.1 | 3.9 | 2.4 | 1.6 | 3.5 | 4.2 | 3.2 | 3.7 | 3.2 |
| Kraken 1.1 | -5.1 | 3.0 | -2.9 | -4.4 | -2.9 | 5.2 | -2.5 | -0.5 | 5.6 | -0.5 |

**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: DEADCODEELIM and BOUNDCHECKELIM 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 recompilations, 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 webpages, 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 pressure 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-

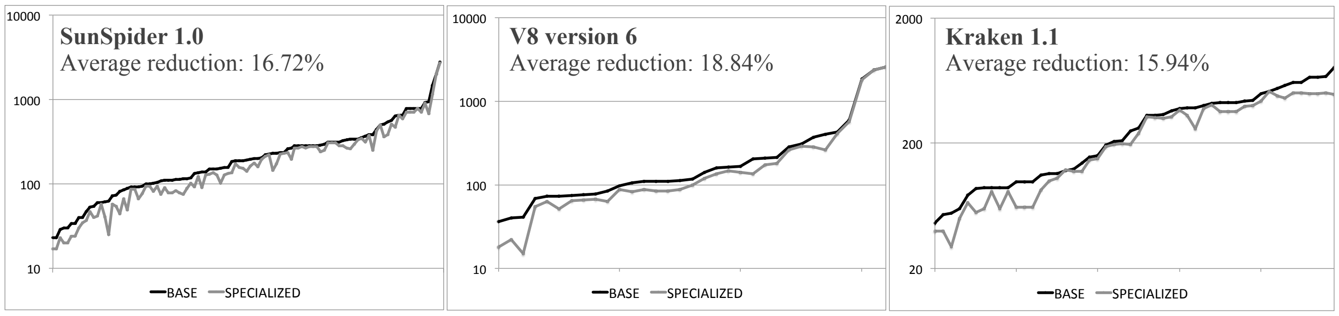


**Figure 10.** Size of generated code (log scale), in number of x86 assembly instructions per function, in three benchmark suites. Each point in the X axis is a function in our test suite. Functions are ordered by the size of the code that IonMonkey produces without our optimizations.

ized 56 functions in SunSpider, 18 were successfully specialized, and 38 had to be deoptimized before program termination. Considering V8, we have specialized 37 functions, 11 of them were successfully specialized, and 26 were deoptimized. For Kraken, these numbers are 38 specialized functions, 14 successful cases, and 24 deoptimizations.

**Impact on number of recompilations.** A function will be recompiled by the JIT engine if an assumption made by the compiler stops being true or more information about the program becomes available. Recompilations are expensive, because they stop the execution of a function to generate a new version of it. In IonMonkey's specific case, recompilations happen, for instance, to update type information or to perform function inlining. Since the value of arguments do not change until at least the function is invoked again, we perform our parameter based specialization even if a function is recompiled. Our approach may increase the number of recompilations of a function, because, in our case, each function will have to be recompiled whenever its is called for a second time with different arguments. We have analyzed how often code is recompiled in our three benchmark suites with and without our runtime value specialization. For SunSpider, the number of compilations of the same function grows by 3.6% when using parameter specialization. This number is 4.35% for V8 and 7.58% for Kraken. Therefore, despite the highly speculative nature of our approach, its drawback, at least in the Firefox benchmarks, is not so big as one could at first expect.

## 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. 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.

**Runtime Value Specialization based on Templates.** Different research groups have proposed to specialize code at runtime with the help of templates built before the program starts running [11, 19, 22]. The template, in this case, is a skeleton of the native code, that will be filled with dynamic information once the program starts running. Consel and

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 use input values passed to the program.

**Type Speculation.** There is a large body of work related to runtime type 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 Psyco, 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 [2].

## Acknowledgments

We thank Nicolas Pierron, 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.

[2] `http://code.google.com/p/jit-value-specialization`

## References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[2] George Almási and David Padua. Majic: compiling matlab for speed and responsiveness. *SIGPLAN Not.*, 37(5):294–303, 2002.

[3] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11. ACM, 1988.

[4] Pericles Rafael Oliveira Alves, Igor Rafael de Assis Costa, Fernando Magno Quintao Pereira, and Eduardo Lage Figueiredo. Parameter based constant propagation. In *Simposio Brasileiro de Linguagens de Programacao*. Sociedade Brasileira de Computacao, 2012.

[5] John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2):97–113, 2003.

[6] Carl Friedrich Bolz, Michael Leuschel, and Armin Rigo. Towards just-in-time partial evaluation of prolog. In *LOPSTR*, pages 158–172. Springer-Verlag, 2010.

[7] Craig Chambers and David Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.

[8] Mason Chang, Bernd Mathiske, Edwin Smith, Avik Chaudhuri, Andreas Gal, Michael Bebenita, Christian Wimmer, and Michael Franz. The impact of optional type information on JIT compilation of dynamically typed languages. *SIGPLAN Not.*, 47(2):13–24, 2011.

[9] Mason Chang, Edwin Smith, Rick Reitmaier, Michael Bebenita, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *VEE*, pages 71–80. ACM, 2009.

[10] Maxime Chevalier-Boisvert, Laurie J. Hendren, and Clark Verbrugge. Optimizing matlab through just-in-time specialization. In *CC*, pages 46–65. Springer, 2010.

[11] Charles Consel and François Noël. A general approach for run-time specialization and its application to c. In *POPL*, pages 145–156. ACM, 1996.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.

[13] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In *POPL*, pages 297–302. ACM, 1984.

[14] Richards G., Lebresne S., Burg B., and J. Vitek. An analysis of the dynamic behavior of javascript programs. In *PLDI*, pages 1–12, 2010.

[15] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, Blake Kaplan, Graydon Hoare, David Mandelin, Boris Zbarsky, Jason Orendorff, Jess Ruderman, Edwin Smith, Rick Reitmair, Mohammad R. Haghighat, Michael Bebenita, Mason Change, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465 – 478. ACM, 2009.

[16] Robert Godwin-Jones. Emerging technologies: New developments in WEB browsing and authoring. *Language Learning and Technology*, 14:9–15, 2010.

[17] Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for javascript. *SIGPLAN Not.*, 47(6):239–250, 2012.

[18] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. In *JAVA*, pages 119–128. ACM, 1999.

[19] David Keppel, Susan J. Eggers, and Robert R. Henry. Evaluating runtime-compiled value-specific optimizations. Technical report, University of Washington, 1993.

[20] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of ACM*, 3(4):184–195, 1960.

[21] Mojtaba Mehrara and Scott Mahlke. Dynamically accelerating client-side web applications through decoupled execution. In *CGO*, pages 74–84. IEEE, 2011.

[22] Francois Noel, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *International Conference on Computer Languages*, pages 132–142, 1998.

[23] Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of javascript benchmarks. In *OOPSLA*, pages 677–694. ACM, 2011.

[24] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *PEPM*, pages 15–26. ACM, 2004.

[25] Raphael Rodrigues, Victor Hugo S. Campos, and Fernando Magno Quintao Pereira. A fast and low-overhead technique to secure programs against integer overflows. In *CGO*, page To Appear. IEEE, 2013.

[26] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *PLDI*, pages 13–22. ACM, 2012.

[27] Marcos Rodrigo Sol Souza, Christophe Guillon, Fernando Magno Quintao Pereira, and Mariza Andrade da Silva Bigonha. Dynamic elimination of overflow tests in a trace compiler. In *CC*, pages 2–21, 2011.

[28] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, pages 125–139. ACM, 2010.

[29] Kai Tian, Eddy Zhang, and Xipeng Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *OOPSLA*, pages 445–462. ACM, 2011.

[30] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.

[31] Shu yu Guo and Jens Palsberg. The essence of compiling with traces. In *POPL*, page to appear. ACM, 2011.