HAL open science

Guided just-in-time specialization

Caio Lima, Junio Cezar, Guilherme Vieira Leobas, Erven Rohou, Fernando Magno Quintão Pereira

▶ **To cite this version:**

Caio Lima, Junio Cezar, Guilherme Vieira Leobas, Erven Rohou, Fernando Magno Quintão Pereira. Guided just-in-time specialization. Science of Computer Programming, 2019, 185, pp.41. 10.1016/j.scico.2019.102318 . hal-02314442

HAL Id: hal-02314442

https://inria.hal.science/hal-02314442

Submitted on 12 Oct 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Guided Just-in-Time Specialization

Caio Lima

Avenida Antônio Carlos, 6627, Belo Horizonte - MG, Brazil

Junio Cezar

Avenida Antônio Carlos, 6627, Belo Horizonte - MG, Brazil

Guilherme Vieira Leobas

Avenida Antônio Carlos, 6627, Belo Horizonte - MG, Brazil

Erven Rohou

Univ Rennes, Inria, CNRS, IRISA

Fernando Magno Quintão Pereira

Avenida Antônio Carlos, 6627, Belo Horizonte - MG, Brazil

Abstract

JavaScript's portability across a vast ecosystem of browsers makes it today a core building block of the web. Yet, building efficient systems in JavaScript is still challenging. Because this language is so dynamic, JavaScript programs provide little information that just-in-time compilers can use to carry out safe optimizations. Motivated by this observation, we propose to guide the JIT compiler in the task of code specialization. To this end, we have augmented the language with an annotation that indicates which function call sites are likely to benefit from specialization. To support the automatic annotation of programs, we have introduced a novel static analysis that identifies profitable specialization points. We have implemented our ideas in JavaScriptCore, the built-in JavaScript engine for WebKit. The addition of guided specialization to this engine required us to change it in several non-trivial ways, which we

Email addresses: caiolima@dcc.ufmg.br (Caio Lima), junio.cezar@dcc.ufmg.br (Junio Cezar), guihermel@dcc.ufmg.br (Guilherme Vieira Leobas), erven.rohou@inria.fr (Erven Rohou), jfernando@dcc.ufmg.br (Fernando Magno Quintão Pereira)

describe in this paper. Such changes let us observe speedups of up to 1.7x on programs present in synthetic benchmarks.
Keywords: Code specialization, Just-in-Time Compilation, Program Annotation, Performance, JavaScript

---

1. Introduction

If we want to improve the efficiency of web-based applications, then we must speed up the execution of JavaScript programs. If in the past JavaScript was considered a mere curiosity among computer hobbyists, today it is one of the Internet's keystones [1]. Every browser of notice supports JavaScript, and almost every well-known website features JavaScript code [2]. Furthermore, JavaScript also fills the role of an assembly argot gluing the web together, considering that it works as a common backend for languages such as Java [3] or Scala [4], and allows the emulation, in browsers, of the LLVM IR [5], and even of whole operating systems[1]. Given this importance, it comes as no surprise the upsurge of attention that JavaScript has received from the industry and the academia in recent years. Much of this attention is focused on the design and implementation of efficient Just-in-Time (JIT) compilers for this programming language [6, 7, 8, 9].

One of the advantages of JIT compilation is *specialization*. Specialization consists in generating code customized to particular types [10, 11, 9], ranges of values [12], constants [13], function targets [14] or architecture [15]. However, even though much has been accomplished by means of specialization, the implementation of this technique on a just-in-time compiler is very difficult. The difficulty stems from a tradeoff: to discover good opportunities for specialization, the JIT compiler must spend time profiling and analyzing the program. Such time is hardly affordable in the fast-paced world of JIT compilation. To mitigate this problem, researchers have proposed different ways to let developers indicate to the runtime environment opportunities of specialization. Examples

[1] `https://bellard.org/jslinux/`

of such approaches range from explicit type annotations [5], to implicit type inference [16], passing through DSLs designed to describe optimizations that include type specialization [17].

In this paper, we defend the thesis that such annotation systems can be extended from types to values. To this effect, we propose the notion of *Guided Just-in-Time Value Specialization*. We describe a minimalistic system of annotations that let developers point to the Just-in-Time compiler which JavaScript functions must be specialized. As we explain in Section 2, specialization happens on a per-argument basis. Different calls of the same function can be specialized for different subsets of its parameters. We implement annotations as comments, following common practice[2]; hence, our extensions to JavaScript are backward compatible. The implementation of an effective guided specialization mechanism required several changes in the JavaScript runtime environment, and led us to design a supporting static analysis; hence, yielding contributions that we summarize below:

- **Insight:** our key contribution is the idea of using annotations to support value specialization. As we explain in Section 2, annotations work like a bridge between the static and the dynamic world; thus, letting the programmer pass information to the runtime environment. Such information is used to enable a kind of partial evaluation of programs [18].
- **Analysis:** the insertion of annotations is a tedious and error-prone task. Thus, we provide two static analyses that, once combined, permit the automatic annotation of programs. The first of them, discussed in Section 3.1, finds *actual* parameters that are likely to remain invariant across multiple executions of the same calling site. The second, presented in Section 3.2, determines *formal* parameters whose specialization tends to be more profitable.
- **Runtime:** code specialization requires, in addition to a specific suite of optimizations, intensive runtime support. This support, the subject of

[2] See *Flow Comments*, at `https://flow.org/blog/2015/02/20/Flow-Comments/`.

Section 4, includes a cache of specialized code, a mechanism to guard specialized code against updates, and a fast verification that checks if the state of objects has changed or not.

We have implemented our ideas in WebKit's JavaScriptCore. To parse annotations, and analyze statically JavaScript sources, we use the Babylon frontend[3]. Section 5 provides a comprehensive evaluation of this implementation. We evaluate our ideas on programs taken from JavaScript benchmarks, such as the JavaScriptCore Benchmark Collection, and on synthetic benchmarks created to demonstrate the power of value specialization. Guided specialization can produce, in some cases, substantial speedups: we have observed, on average, speedups of 1.12x on synthetic benchmarks, with a maximum performance boost of 1.7x. Although our average gains seem small at first, we recall the fact that we are comparing against an industrial-strength compiler, with all its optimizations enabled. Furthermore, our measurements include time spent on code specialization.

2. Overview

We use the notion of *specialization* that Definition 0.1 states. We define specialization as a form of partial evaluation [18] restricted to functions. As the reader might wonder, this is a rather narrow notion. Nevertheless, it lets us focus on the key insights of this paper:

- **Selective specialization**: contrary to unrestricted value specialization [13], we generate functions specialized to only some of their actual parameters, at specific calling sites.
- **Static Analyses**: we determine, automatically, the calling sites that are likely to benefit more from specialization, and rule out those deemed unprofitable.
- **Runtime guards**: objects contain meta-information that track where

[3]Babylon is available at `https://github.com/babel/babylon`.

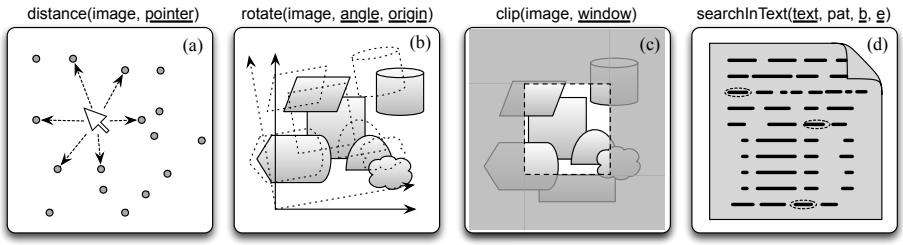

Figure 1: Examples of value specialization. The value that will be specialized is underlined at each function call; each function tends to be invoked many times with the same underlined arguments. (a) Compute the distance from a graphic object (image) to the mouse pointer. (b) Rotate an image according to an angle and origin vector. (c) Compute the part of an image that is within a window. (d) Pattern matching.

they have been updated. This information lets us deoptimize specialized values that have been updated.

Definition 0.1 (Function Specialization)**.** Let f be a function that receives a set F of formal parameters. Let $I_{static} \subseteq F$ be a subset of parameters whose value is *known*. We say that f' is a specialized version of f for I_{static} if f' results from the partial evaluation of f on I_{static}.

As Definition 0.1 states, this paper focuses on the specialization of functions, given the arguments they receive at different call sites. We opted to specialize function calls, instead of general program regions because: (i) this approach is easier to implement in a JIT compiler; and (ii) it is widely applicable. Figure 1 illustrates some of these applications for image processing functions, and a pattern matching algorithm, further detailed in Figure 2. Function searchInText, in Figure 2, receives an array of objects (text), a query (q), plus integers (begin) and (end). It finds elements between the begin-th and the end-th index of text that equals q. Function searchInText could be specialized for any subset of its four arguments; however, some subsets are more profitable. In particular, line 17 of Figure 2 reveals that the actual instances of begin and end are loop invariant. Hence, once specialized for these two arguments, searchInText would be

```
1  function searchInText(text, q, begin, end) {
2    let foundPos = [];
3    if (begin < 0 || begin > text.length)
4      return foundPos;
5    if (end < begin || end >= text.length)
6      return foundPos;
7    for (let i = begin; i < end; i++)
8      if (text[i] == q)
9        foundPos.push(i);
10   return foundPos;
11 }
12 let wordList = document.getElementById("listOfWords");
13 let b = document.getElementById("beginOfText"); /*b==1*/
14 let e = document.getElementById("endOfText");    /*e==4*/
15 let txt = ["prove", "all", "things", "hold", "fast", "that", /*...*/];
16 for (let i = 0; i < wordList.length; i++) {
17   let findPos = searchInText(txt, wordList[i], b, e);
18   print("Found \"" + wordList[i] + "\" in following positions:");
19   for (let pos of findPos)
20     print(findPos);
21 }
```

Figure 2: Example that we shall use to illustrate the benefits of guided code specialization techniques.

used many times: once for each iteration of the loop at lines 16-21 of Figure 2. Example 1 illustrates this scenario.

Example 1. Figure 3 (a) shows the code that could be produced out of the specialization of arguments begin and end used in function searchInText (Fig. 2). In this example, we let begin $= 1$ and end $= 4$, as it will be the case, if we specialize the call of searchInText at line 17 of Figure 2.

```
1  function searchInText(text, q, begin, end) {
2    let foundPos = [];
3    if ("prove" == q) { foundPos.push(1); }
4    if ("all" == q) { foundPos.push(2); }
5    if ("things" == q) { foundPos.push(3); }
6    return foundPos;
7  }
```

(a)

```
12 ...
13 for (let i = 0; i < wordList.length; i++) {
14   let findPos = searchInText(  /* #specialize(0, 2, 3) */
15                                 txt, wordList[i], b, e);
16   print("Found \"" + wordList[i] + "\" in following positions:");
17   for (let pos of findPos) print(findPos);
18 }
```

(b)

Figure 3: (a) Specialized version of searchInText. (b) The specialize pragma used in a program.

The Speculative Nature of Specialization. Just-in-time function specialization is a form of *speculation*. The gamble, in this case, lays in the compiler's assumption that some arguments are likely to remain invariant, regardless of how many times the target function is called. If the target function is called with actual parameters holding values different than those observed during specialization,

the JIT compiler will pay a time fee for that speculation. Continuing with our example, the specialized version of searchInText, seen in Figure 3 (a), only works for begin=1 and end=4. If searchInText is invoked with different arguments, then either the JIT compiler will produce new code for it, or we must fall back into interpretation mode. Therefore, blind specialization, à la Costa *et al.* [13] might, sometimes, lead to performance degradation, instead of producing speedups.

Guided Specialization. To avoid performance degradation, we believe that good specialization opportunities can be discovered statically, be it through human intervention or automatic static analyses. We call this combination of static analysis and dynamic optimization *Guided Specialization.* In this paper, we define a simple annotation, the specialize pragma, to guide the JIT compiler. This pragma can be inserted by the developer, or inferred by static analyses. Figure 3 (b) shows how the specialize pragma is used to guide the specialization discussed in Example 1. In our current implementation of guided specialization, the specialize pragma is written into a program as a JavaScript comment. This comment must appear as part of a function call, within the list of actual parameters of the function that shall be specialized. The specialize directive is parameterized by a list of integers. Each integer denotes an actual parameter of the function call that must be specialized. The correspondence between the arguments of specialize and the formal parameters of the function that will be optimized is positional. The first position is indexed by zero.

Example 2. The call of searchInText, at line 14 of Figure 3 (b) has been marked for specialization. This call contains four arguments: txt, wordList[i], b and e. This invocation of seachInText must be specialized for its first, third and fourth arguments, which correspond to indices 0, 2 and 3 in the specialize directive.

Automatic Guided Specialization. The specialize pragma can be used directly by developers. However, this pragma can also be inserted into programs by a static analysis tool, without any human intervention. The use of static analysis to annotate code is not an uncommon practice. For instance, Campos *et al.* [19]

have designed a compiler front-end that inserts the restrict keyword automatically in C programs. DawnCC [20] and the TaskMiner [21] use static analyses to insert OpenMP annotations into C programs. Pominville *et al.* [22] annotate Java class files to help the JIT compiler to eliminate array bound checks. Along similar lines, we have designed two static analyses that, combined, find calling sites that can be profitably annotated. The first of these static analyses, which we shall explain in Section 3.1, finds calling sites in which some of the arguments are *unlikely* to change. Example 3 shows this analysis in action.

Example 3. The invocation of function searchInText at line 17 of Figure 2 contains two arguments which will never change: its third and fourth parameters. That calling site also contains an argument that is unlikely to change: the first. It may still change, for instance, due to destructive updates in objects that alias txt. If that happens, then deoptimization takes place.

The second static analysis, the subject of Section 3.2, finds functions that can be "profitably" optimized, given that we know the value of some arguments. We say that specialization over some particular argument is profitable if this optimization is likely to "influence" a level of code above a certain threshold. Usually it is not possible to estimate this influence, unless we carry out the actual partial evaluation of the function. However, we can approximate it via some form of *information flow analysis* [23]: the more statements an argument taints, the more likely we are to obtain more extensive optimizations.

Runtime Guards. We emphasize that guided specialization is still a speculative operation. Aliasing, for instance, might hide changes in arguments deemed unlikely to change. For instance, side effects in statement foundPos.push, in Figure 3 (a) could change objects text, begin or end; hence, rendering our specialized code invalid. In face of such destructive updates, the specialized function must be discarded, and more general code must be produced for it. This process is called *deoptimization*. We do not solve alias analysis statically. Rather, we augment objects with *allocation stamps*, as we explain in Section 4.1. An allocation stamp is a kind of meta-information attached to each object, which identifies

where that object has been last updated. If a function f, specialized to a given object o_1, receives an object o_2, such that o_1 and o_2 are updated at different sites, then f is deoptimized. However, we do not trash the specialized function. We keep it in a cache, so that we can still reuse it, in case that is possible in future invocations. Another interesting detail of our implementation is that it does not impose any overhead onto objects that have not been specialized. Specialized objects are kept in read-only memory. Attempts to update them lead to exceptions that we capture. In this way, we find which objects need allocation stamps, and which objects do not need them. Such implementation details are covered in Section 4.3.

3. Guided Value Specialization

In this section, we describe how we use static analyses to discover opportunities for value specialization, and how we generate specialized code. Figure 4 provides an overview of this process. We use two static analyses, described in Sections 3.1 and 3.2. They happen at the source code level, before the execution of the JavaScript program starts. They receive as input a JavaScript program, and produce as output an annotated version of that program. Annotations are formed, exclusively, by the `specialize` pragma, which appears within the list of arguments of function calls deemed profitable for specialization. These two static analyses do not require any change in the JavaScript language, or in the language runtime, to be effectively used.

However, the implementation of guided specialization requires changes in the JavaScript interpreter, and in the Just-in-Time compiler. Such changes, which we explain in Section 4 allow the JavaScript runtime environment to keep track of the sites where objects are modified. Tracking object modifications lets us implement less conservative, yet safe, object specialization. We emphasize that specialization happens *per function call*, not *per function*. Each function call leads to a different specialization. Specialized versions of functions are kept in a cache, to be reused whenever possible.

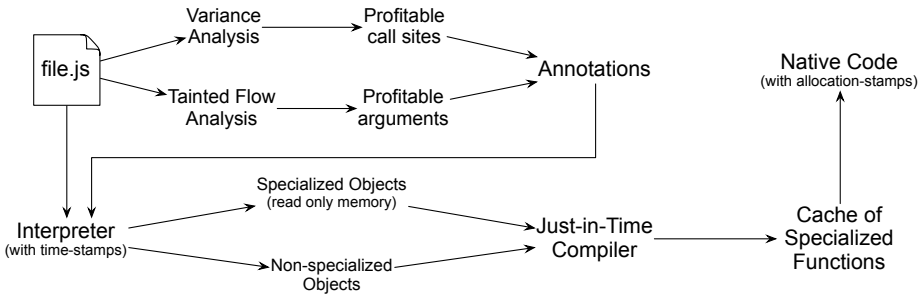

Figure 4: Overview of the value specialization process that we propose in this paper.

3.1. Variance Analysis

Variance analysis [24] is a static analysis technique that determines which program variables remain invariant within loops. We use it to find actual parameters of functions that are likely to be invariant across multiple invocations of those functions. These actual parameters are good candidates for specialization. Our implementation of this analysis differs from the typical textbook implementation, e.g., à la Appel and Palsberg [24] in two ways. Firstly, it is global: our implementation works for the entire function, instead of on a per-loop basis. This distinction is important, because it changes substantially the implementation of variance analysis, and comes from our motivation to use this analysis. Whereas variance analysis has been typically used to hoist code outside loops, we use it to enable value specialization. Secondly, we have implemented variance analysis directly on the program's abstract syntax tree (AST) – it runs on JavaScript sources – instead of on an intermediate representation based on three-address code, as it is commonly reported in the literature [25, 24].

Variance analysis is parameterized by a *reaching-definitions* function R. $R(\ell)$ is the set of definitions that reach label ℓ. Because reaching definitions is a standard concept in compiler optimization, we shall omit its formal definition. The result of variance analysis is a set V of triples $(\ell : v, \ell_w)$. The triple $(\ell : v, \ell_w)$ indicates that variable v, defined at label ℓ, is invariant in the loop

$$\texttt{[Prp1]}\quad \frac{\exists u \in U \qquad \exists(\ell_1, u) \in D \qquad \exists(\ell_2, u) \in D \qquad \ell_1 \neq \ell_2 \qquad \ell_w : \texttt{loop}(e, c) \in P \qquad \ell_1 \in c}{\text{Property}_1(P, \ell_w, U, D) \text{ is true}}$$

$$\texttt{[Prp2]}\quad \frac{\exists u \in U \qquad \exists(\ell_x, u) \in D \qquad \exists(\ell_x : u, \ell_y) \in V}{\text{Property}_2(U, D, V) \text{ is true}}$$

Figure 5: The sources of variance information.

```
ℓ1: x = 0;
ℓ2: loop(x < N) {
 ℓ3: x = x + 1;
}
```
(a)

```
ℓ1: x = 0;
ℓ2: k = 1;
ℓ3: loop(x < N) {
 ℓ4: y = x;
 ℓ5: x = k;
 ℓ6: x = y + x;
}
```
(b)

```
ℓ1: x = 1;
ℓ2: loop(x < N - 1) {
 ℓ3: y = x + 1;   [at ℓ2]
 ℓ4: loop(y < N) {
  ℓ5: y = y + x;  [at ℓ2,ℓ4]
 }
 ℓ6: x = x + 1;   [at ℓ2]
}
```
(c)

```
ℓ1: loop(x < N) {
 ℓ3: x = foo();
}
```
(d)

Figure 6: Examples of usage of the variance analysis. Black definitions are deemed variant by Property 1 ([Prp1] in Figure 5); grey definitions are marked variant by Property 2 ([Prp2] in Figure 5). (a) Variable x is reached by two definitions, and one of them comes from within the loop at ℓ_2. (b) Variable y is variant within ℓ_3, because ℓ_4 is reached by two definitions of x, and one of them comes from inside the loop. In contrast, x is not variant at ℓ_6, because it is reached by only one invariant definition. Finally, x at ℓ_6 is variant due to Property 2, given that y is variant. (c) Our variance analysis produces results per loops. Hence, the same variable can be variant in some loops, and invariant at others. Such is the case of x: it is variant in ℓ_2, but not in ℓ_4. (d) Variable x is variant, because it is defined by a function call, whose result might vary each time it is invoked.

located at label ℓ_w.

Variance information is passed forwardly via the verification of two properties: "**P1**" and "**P2**" below:

P1 : A variable v, defined by $\ell : v = e$ or $\ell : v.y = e$, is variant if:

1. ℓ is located inside a loop at ℓ_w,
2. there exists a variable $u \in e$,
3. at least two definitions of u reach ℓ, and
4. one of these definitions comes from within the loop ℓ_w.

P2 : a variable v is variant if its definition uses some variant variable. For instance, if u is variant, and v is created by the instruction $v = u+1$, then v is also variant.

Notice that **P1** distinguishes assignments into scalar and object values. An assignment $x.y := e$ has a double effect: it sets the invariance (or its absence) on the names x and y. Thus, the assignment $x.y := e$, for the purpose of variance analysis, is equivalent to two assignments, e.g.: $y := e$ and $x := e$.

Example 4. Figure 6 shows the variance analysis applied onto different programs. Figure 6 (d) illustrates how variance analysis propagates information throughout function calls: the outcome of a function call invoked within a loop is marked variant at that loop, because the result of the call might vary at each iteration of the loop.

Our implementation of variance analysis is guaranteed to terminate, and it runs in time linear on the number of dependences in the program. A dependence between variables u and v exist whenever the program contains an instruction that defines v and uses u. Termination is ensured by a simple argument: a variable is either variant or invariant. The analysis starts assuming that every variable is invariant, except for those that bear Property **P1**. Once Property **P2** is used to make a variable v variant, it will never again become invariant.

3.2. Profitability Analysis

As we shall demonstrate in Section 5, it is not difficult to find function calls with invariant actual parameters. However, reckless specialization might easily lead to performance regressions, because it is often the case that arguments are only used a few times within the body of functions. The replacement of such arguments with constant values does not lead to much performance improvement. On the contrary, the overhead of having to recompile functions, in case deoptimization takes place, usually overshadows any possible gains we could expect. Thus, guided value specialization requires some effort to identify argu-

ments that are likely to lead to real performance improvement, once specialized. This is the goal of the so-called *Profitability Analysis*.

What we call Profitability Analysis is a mix of tainted-flow analysis [23] and classic constant propagation – without the effective replacement of instructions with constants. The goal of this analysis is to find every statement that can be completely resolved at the time of code generation, in case we decide to specialize a function. The profitability analysis uses a simple lattice, formed by the partial ordering unknown $<$ profitable $<$ unprofitable. It works per function. The abstract state of variables is initialized according to the following properties:

P3 : Variables marked as invariant by the analysis of Section 3.1 are profitable.

P4 : Variables marked as variant are unprofitable.

P5 : Other variables are unknown.

Information is then propagated according to the following property:

P6 : If every variable used in the definition of variable v is profitable, then v is marked as profitable.

Our implementation of the profitability analysis is parameterized by a function A, which maps names to aliases. For example, $A(x) = \{a, b, c\}$, denotes the possibility of a, b, c and x being pointers to the same location. Thus, the abstract state of a variable v is the same as the abstract state of all its aliases.

Like variance analysis, profitability analysis terminates without the need of any special widening operator. We can see this analysis as a way to propagate the tainted state: "unprofitable". If a definition uses an unprofitable variable, then it is marked unprofitable. Once a variable is marked unprofitable, its state never changes. Thus, the evaluation is monotonic. Because it uses a lattice of finite height, it terminates after each variable is visited at most twice.

Computing Profits. Once we have marked every variable in the function as profitable or not, we compute the profit of specializing that function. There

exist different cost-models that we could adopt at this stage. We have settled for a simple one: the spilling model advocated by Appel and George in their description of the Iterated Register Coalescing algorithm [26]. Following this cost model, we define the profit of specializing a function F with the formula below. In this formula, n is c's nesting depth, D is the cost to run c once, and L is the number of times a loop is expected to run. The nesting depth of a statement is the number of loops surrounding that statement. And, to estimate L, we simply assume that each loop runs a constant number of times. In our current implementation, $L = 10$.

$$P(F) = \sum_{c \in F} L \times D^n \text{ if } c \text{ only uses "profitable" definitions} \tag{1}$$

If the profit of specializing a function call exceeds a given threshold, then we annotate that call with the #specialize pragma. We are using a simplistic cost model, which relies on several magic-numbers, such as D and L. We could, in principle, use more elaborate cost models, such as those adopted in static profiling techniques [27, 28]. Our notion of guided JIT specialization does not prevent this kind of fiddling; on the contrary, we believe that more complex cost-models can be integrated into our framework seamlessly. For the moment, we claim that this type of exploration is orthogonal to our ideas, and we leave it open as future work.

Example 5. Profitability analysis fits the same distributive framework [29, Sec.2] as constant propagation, but these analyses are different. Figure 7 illustrates the difference. Variable foundPos, used at line 7 would be considered non-constant by constant propagation. Profitability analysis sets this use as profitable, because all the definitions of foundPos reaching line 7 are set as profitable.

4. The Runtime Environment

Our implementation of guided specialization exists onto two pillars: allocation stamps and specialization caches. In the rest of this section we describe

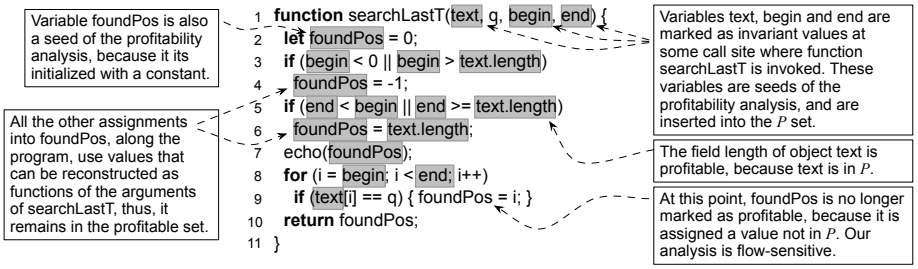

Figure 7: The profitability analysis in action. Statements considered profitable are marked with grey boxes.

these concepts, starting with the former.

4.1. Allocation Stamps

Allocation stamps are meta-information that we append to JavaScript objects, in order to track the site where these objects have been last modified. At the semantic level, the main difference between our version of JavaScript, and the original language (as formally described by Maffeis *et al.* [30] or Park *et al.* [31]) is the inclusion of this notion, which Definition 5.1 formalizes.

Definition 5.1. An allocation stamp is a meta-information associated with every object created during the execution of a program. This meta-information identifies, unambiguously, the program site where each object was last modified. If r is a reference to an object u modified at ℓ, then at ℓ we update u's stamp with an assignment $r.stamp = \ell$.

Allocation stamps track *one level of field sensitiveness*. Thus, if s is the stamp associated with object u, we update it if one out of these two situations takes place:

1. u is initialized, i.e., is assigned a new allocation site;

2. $u.f$ is modified, where f is an attribute of u.

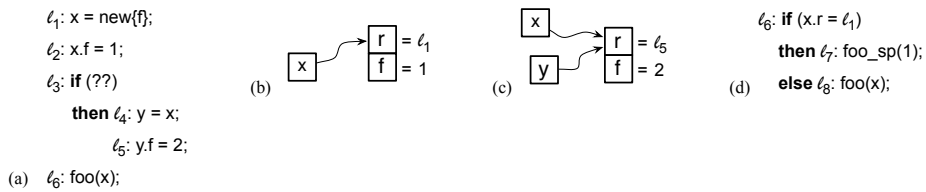

Figure 8: How allocation stamps are used. (a) Program that might contain aliasing between references x and y at the calling site ℓ_6. (b) State of the store environment after instruction ℓ_2 is executed. (c) Store state after instructions ℓ_4 and ℓ_5 run. Object r receives a new allocation stamp at ℓ_5. (d) Invocation site ℓ_6, after specialization. foo_sp is a version of foo specialized to the value 1.

This semantics shows that allocation stamps track *shallow modifications*. As an example, the allocation stamp of object u does not change if $u.v.w$ is modified.

Allocation stamps let us carry out non-conservative value specialization. The variance analysis of Section 3.1 finds variables that are likely to remain invariant inside loops. However, the state of these variables might change, due to aliasing. If we let aliasing limit the reach of our static analysis, we would not find many opportunities for value specialization. Allocation stamps let our variance analysis assume that aliasing never occurs. If the state of an object changes at the calling site where it has been specialized, then a check on the allocation stamp is enough to detect this fact. Example 6 clarifies this modus operandi.

Example 6. Figure 8 shows a program where the possibility of aliasing would force a static analysis to assume that the invocation of function foo cannot be specialized. Variance analysis (Sec. 3.1) assumes the absence of aliasing. To ensure correctness, we insert a guard at the specialization site. This guard, seen in part (d) of Figure 8, checks if the allocation stamp of the concrete object matches the allocation site that variance analysis has assigned to that specialized value.

Allocation Stamps and Inline Caching. Inline caching [32, 14] is a form of *type specialization*, because the same cache works for any object that is an instance

```
function Obj(v) {
  this.t = 'T';
  this.i = v;
(a) this.get = function get(x) {
    return this.i + x;
  }
}
```

```
ℓ1: o = {i: 42}
(b)
```

```
ℓ2: o.get(x)
(c)
```

```
if (o.t == 'T')
  o.i + x;
(d)
```

```
if (o.r == 'ℓ1')
  42 + x;
(e)
```

Figure 9: Type specialization based on inline caching versus value specialization. (a) JavaScript object with a type property T, and a field value i. (b) Example of object creation at an allocation site ℓ_1. (c) Invocation of function get over an object o, instance of T. (d) Specialization based on inline caching. (e) Value specialization.

of T. We specialize a call site for some value v stored in the field of an object created at location ℓ. Thus, our form of specialization is more restrictive: it works for values instead of types. However, it is also more aggressive: we can not only generate code specialized for a certain type, but also for a certain value. Hence, we support, like inline caches, the inlining of method calls, but, unlike it, we also support value-based optimizations, such as constant propagation. Figure 9 illustrates these differences. One of the more effective sources of speedup we have observed in our experiments was, indeed, the replacement of side-effect-free *getters* with constants, like we show in Figure 9.

Reducing the overhead of allocation stamps. Allocation stamps impose some overhead onto the runtime environment, because they must be updated whenever the object is updated. However, it is possible to reduce considerably this overhead, as long as we are willing to let deoptimizations happen more often. This is the approach that we have adopted in our implementation of guided specialization. Our implementation ensures two properties:

- there is almost no overhead imposed onto non-specialized objects;
- once deoptimized, a specialized object behaves like an ordinary object; thus, not suffering further overheads.

To ensure these two properties, we resort to the following stratagem: we mark memory pages that hold specialized objects as read-only. If a write operation

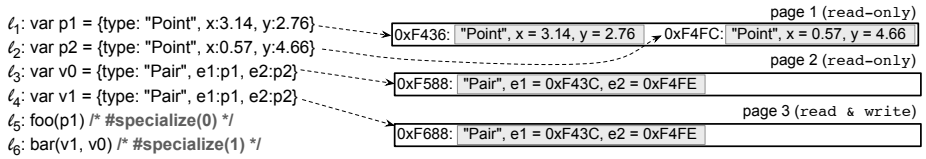

Figure 10: Page 1 is marked `read-only` because it contains object `p1`, which is specialized due to the call at ℓ_5. Object `p2` shares the same page, by chance. Attempts to update it will update also `p1`'s stamp. Page 2 also contains a specialized object: `v0`; hence, it is marked `read-only`. Page 3, which does not contain any specialized object, is marked `read & write`.

targeting such a page takes place, then a segmentation fault ensues. We trap this signal, and update the allocation stamps of all the specialized objects located in that page. Our technique has advantages and disadvantages, which we shall evaluate in Section 5.1. On the one hand, it bears no impact whatsoever on programs that do not contain any object specialization. On the other hand, updating –and thus decompilation– happens even when write operations target non-specialized objects stored in a page that contains specialized objects. Furthermore, we must bear in mind that signal handling is a relatively heavy operation; thus, updating of specialized objects has a high computational runtime cost. Such heavy cost puts even more importance on the good quality of our static analyses, as they mitigate this overhead. Figure 10 illustrates this idea.

4.2. Specialization Caches

We keep different versions of a specialized function into a *specialized code cache*. This cache is a table that associates tuples $(f, v_1, \ldots, v_n)$ with code. Code, in this case, is an implementation of f, specialized to the values $v_1, \ldots, v_n$. When the JIT compiler finds a call site ℓ of a function f, it checks if f is "specializable". This is true if some call site of f – possibly different than ℓ – was annotated with the `specialize` pragma. If that is the case, then the JIT compiler checks if the actual arguments of f, at ℓ, have an entry in the cache.

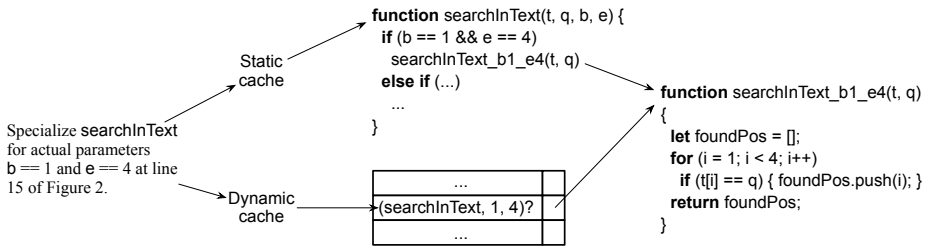

Figure 11: The static and the dynamic code caches.

When such is the case, we do not generate code for f, but reuse its cached implementation.

We have experimented with two different implementations of the code cache: the *dynamic cache* and the *static cache*. In the dynamic version, the JIT performs the search for $(f, v_1, \ldots, v_n)$ in the code cache. That is to say: we have a generic routine that queries the cache. In the static version, we augment each specialized function with a prolog, which queries directly its actual arguments. Our experiments indicate that the static version is more efficient. If we run benchmarks with the cache enabled, but without performing any optimization, to gauge the cache's overhead, then we observe slowdowns of up to 30% with the dynamic implementation in the benchmark suite synthetic benchmarks. The overhead of the static cache is never greater than 4% on the same benchmarks.

Figure 11 illustrates the difference between these two approaches of code caching. Our current implementation of the code cache in JavaScriptCore keeps four entries for each function. If the program has more than four call sites, for the same function, marked with the specialize pragma, we might have to modify this function's prolog. Prolog modification does not require whole code reconstruction, because we are simply replacing constants in binary code. We replace the oldest entry with the new one.

4.3. Implementation Details

Thus far, we have kept our presentation mostly abstract; however, our ideas are implemented in JavaScriptCore (JSC), an industrial-quality runtime engine for the JavaScript programming language. Adapting our ideas to run in JSC involved major changes into its design. In this section we describe some of these changes. However, before, we provide some explanation on the internals of JSC itself, so that the reader can better understand how we have modified it.

JavaScriptCore in a Nutshell. JavaScriptCore follows the state machine presented in Figure 12. In the first stage of the execution of a JavaScript program, JSC parses the JavaScript source code and translates it into a low-level representation called "the bytecode format". Bytecodes are then interpreted. During this interpretation, the program is profiled. Profiling gives JSC the runtime information that will be used by different layers of JIT compilation, once the time to produce machine code arrives. Among such information, we count *execution thresholds*: counters that measure how often branches are taken, and how often functions are invoked. Once different thresholds are reached, different versions of the Just-in-Time compiler are invoked. Each of these versions applies on the target program more extensive optimizations. JavaScriptCore uses the following JIT tiers:

- **Baseline**: this is the most basic JIT layer. Its goal is to generate code

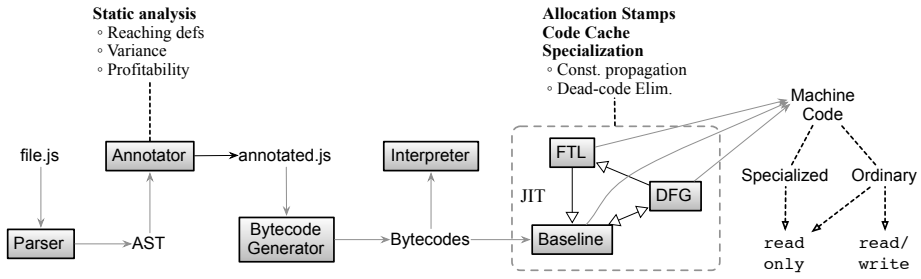

Figure 12: JavaScriptCore's compilation pipeline after our interventions on its implementation.

as fast as possible. The baseline compiler is invoked whenever the same function is called six times, or some statement is interpreted 100 times. At this stage, no optimizations are carried out.

- **DFG**: this is JSC's second JIT layer, and it is invoked after some baseline-compiled function is invoked 66 times, or some baseline-compiled statement is invoked 1,000 times. At this stage, we observe the first optimizations being applied onto the target program. In this process, the program is converted into a representation more suitable to compiler optimizations, the Data-Flow Graph (DFG). Among the optimizations used at this point, we count dead-code elimination, strength reduction, constant propagation and common subexpression elimination.
- **FTL**: this stage, popularly known as the *Faster-Than-Light* JIT[4], is JSC's last optimization phase. It is invoked if some counter reaches a threshold of 100,000 executions. At this point, the JIT compiler applies onto the target program several optimizations, including register allocation via the classic Iterated Register Coalescing [26] algorithm.

Our Interventions. To equip JSC with support to guided specialization, we adapted it, so that the interpreter's parser could support annotations. To this end, we have modified the JavaScript parser to propagate annotations to the DFG and FTL optimization layers. Our code specialization machinery runs at these two levels: DFG and FTL. Once the JIT receives a function augmented with meta-information describing some particular specialization policy, our code specializer is invoked. The specializer performs two actions of notice: the replacement of references by constants, and the generation of the prolog to implement the static code cache. Once we replace references by constants, the optimizations already in place in JavaScriptCore perform constant propagation, value numbering and dead-code elimination; thus, yielding optimized binaries. The implementation of allocation stamps required us to modify JSC's memory

[4] `https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/`

manager. Changes included the addition of an extra field, the stamp, in the table of meta-information associated with specialized objects, and modifications in the allocation strategy, as illustrated in Figure 10.

The Implementation of the Static Analyses. In typical Web browsing, JavaScript files are downloaded from the server-side and then executed. For maximum performance, we envision the application of our static analyses and the annotation of JavaScript code at the server-side, before download takes place. Our static analyses run at source-code level; hence, we analyze Abstract Syntax Trees. We have implemented the static analyses using the Babylon Framework[5]. The benefit of running static analyses at the source-code level is the fact that we can rely on high-level information, like the semantics of JavaScript constructs and scoping rules to improve static results. The downside of this approach is the fact that we lose the opportunity to use the Static Single Assignment form [33], which would have simplified the implementation of our analyses. We emphasize that our static analyses can be used in combination with human-made annotations. In other words, if the programmer annotates some function call with the specialize pragma, such directive will not be removed by the static analyses, even if the function call is deemed unprofitable. Currently, our implementation does not give warnings about non-profitable annotations inserted by human developers. In other words, if we apply static analysis on an already-annotated JavaScript file, then some new annotations might be created, but old annotations are never removed.

5. Evaluation

This section demonstrates, with experimental data, that guided JIT compilation is effective and useful. To this end, we shall seek to provide answers to a few research questions:

[5] `https://github.com/babel/babylon`

- **RQ1**: can guided specialization increase the performance of JavaScript programs?
- **RQ2**: what is the impact of guided specialization on the size of the native code?
- **RQ3**: what is the impact of the profitability threshold onto the quality of the code produced?
- **RQ4**: what is the overhead of the allocation stamps that we have introduced in Section 4.1?
- **RQ5**: how does guided specialization fare when compared to traditional specialization [13]?

Before we dive into the task of addressing each of these research questions, we describe our experimental setup: the underlying JavaScript engine that we use, our runtime environment, our benchmarks and our timing methodology.

- **Runtime Environment.** All our experiments have been performed on an Intel Core i5-2500S CPU with 4 cores running at 2.7GHz. The operating system that we use is macOS Sierra version 10.12.1 (16B2659). Memory usage experiments have been performed in an Intel Core i7-3770 CPU with 8 cores running at 3.40GHz. The operating System that we use is Linux v4.10.0-33-generic 16.04.1-Ubuntu SMP. Virtual pages have 4K. The version of JavaScriptCore – the Webkit's VM – that we have used to implement our approach (and also as a baseline) is r220016
- **Benchmarks.** To test our implementation, we have used two benchmark suites: JavaScriptCore's benchmark collection (JBC) and a synthetic suite with seven benchmarks that let us illustrate the power of value specialization. JBC contains 582 JavaScript programs. Each of them tests some aspect of JavaScriptCore's runtime engine. We use this collection when answering some of the research questions, because the larger number of programs lets us run experiments that the small number of samples in the synthetic suite would yield meaningless. An example is the histogram that reports code size increase, which we analyze in Section 5.3. Nevertheless, most of the programs in JBC run for a very short time, which makes it dif-

ficult to draw conclusions about their performance. Thus, we have added seven new benchmarks to this collection. Each benchmark consists of one or more hot functions, which call methods from the JavaScriptCore standard library, such as `Math.log`, `Math.random`, `Array.get`, `Array.push`, `String.length`, etc. These programs cannot be easily optimized by the current implementation of JavaScriptCore, whereas the techniques in this paper can boost their runtime. Figure 13 describes each one of these benchmarks.

- **Timing methodology.** Runtime numbers are the average of 30 executions, including the time of warmup and compilation. We do not control CPU features like turbo boost or hyper-threading. We measure runtime invoking a specific script using JSC command line interface, i. e., each run of a benchmark is done in a different process. We do not perform any warm-up before running each benchmark; because some of our interventions happen in the JavaScript interpreter. In particular, the management of object stamps happens not only in the code produced by the Just-in-Time compiler, but also during interpretation. We only report numbers for benchmarks that give us *definite changes*. JCS's testing framework provides us with error margins and standard deviations for each comparison between different versions of its JIT compiler. Given two sets of samples, S_1 and S_2, they are only considered different if their p-value is lower than 0.05[6], following Student's Test [34].

[6]The smaller the p-value, the higher the significance of the result, because the p-value is the probability that we could observe our speedups in a setup in which the null-hypothesis (there should be no speedup) is true. Following the default testing infrastructure used in JavaScriptCore, we consider statistically significant every experiment whose p-value is lower than $\alpha = 0.05$. However, if the reader is interested in adopting lower significance levels, then we include explicitly the p-value computed at every experiment.

#L	Benchmark	Description
8	`get-base-log`	This benchmark shows how guided specialization allows us to constant-fold one call of `Math.log`, while keeping another, less profitable, unspecialized.
80	`image-clipping`	The operation of clipping just part of a image given a window. This benchmark stresses geometric operations, mainly the verification if a given point is out-of-bounds of a rectangular area. The area is an object that can be specialized.
19	`image-rotation`	This operation rotates 2D points given an angle. It stresses math operations to rotate objects in a scene. The angle is fixed, and can be specialized.
59	`point-find-closest`	Operation to find the point that is the closest to a given query point. This benchmark stresses the effectiveness of object value specialization, since each point is an object.
73	`rabin-karp`	One implementation of a string matching algorithm. This benchmark shows how string attributes can be selectively specialized. It is profitable to specialize the searched pattern, but not the source text.
42	`simple-time-format`	Simplified algorithm to format time input given in seconds. This benchmark illustrates the benefit of specializing primitive types.
28	`simple-vm-case`	Simplified implementation of a VM with only 2 operations (OP_INC and OP_MIN). This benchmark contains only the function that interprets these two operations. These functions can be selectively specialized for each operation.

Figure 13: Overview of synthetic benchmarks. **#L**: number of uncommented lines of code. Each benchmark and the harness used to run it is available at `https://github.com/caiolima/synthetic-benchmarks`.

5.1. RQ1 – Effectiveness

The effectiveness of our approach is measured by the amount of speedup that we can bring on top of the original implementation of JavaScriptCore's JIT compiler. The original version of this compiler does not carry out value specialization, but it uses three different optimization levels (as described in Section 4.3) to produce high-quality code. We keep all these levels, and augment them with the ability to produce code that is specialized to particular values. Figure 14 shows how our new version of JavaScriptCore's Just-in-Time compiler does when compared against the baseline implementation. We report values of two configurations: the first only specializes numeric and boolean types, which we shall call *primitive values*; the other specializes JavaScript objects in general. We have used a profitability threshold of 70 cycles.

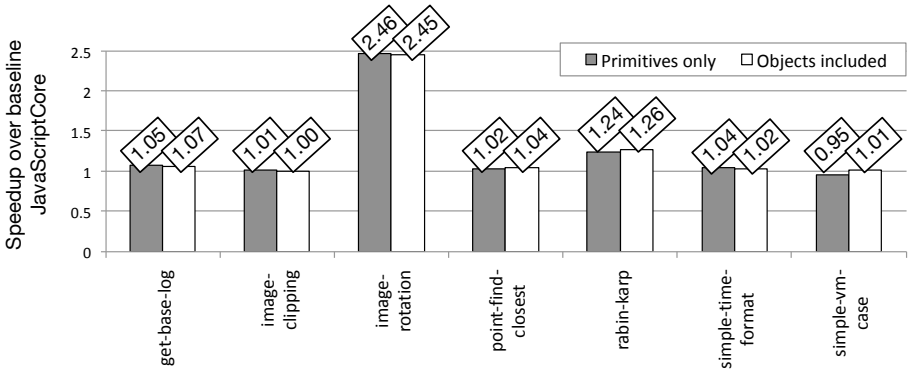

Figure 14: Runtime difference between guided specialization and baseline JavaScriptCore with all its optimizations enabled. Each bar represents a different benchmark. Y-axis shows time difference between original and optimized code (in number of times). The higher the bar, the better. Bars that are below 1 mean slowdowns. Numbers above bars show speedups (in number of times).

Figure 15 shows the absolute runtimes of the seven benchmarks that we have used to produce the bars seen in Figure 14. In Figure 15, we show above the optimized runtimes the p-value derived from a statistical test applied onto the original implementation of JavaScriptCore and its optimized version. Considering significant the experiments with a p-value under 0.01, we observed that the setup that specializes only primitive values achieved speedups in 3 benchmarks, out of 7. The setup that also optimizes objects got significant speedups in 4 out of 7 benchmarks. Slowdowns are due to the cost of keeping allocation stamps, and to deoptimizations.

Our highest speedup was 2.4x on image-rotation. This performance boost is due to the specialization of two calls to `Math.cos`, and two calls to `Math.sin`. The only apparent slowdown was 0.95x on simple-vm-case with specialization of primitive types. However, in this case, the p-value of 0.48 (see Fig. 15) does not lets us deem this slow down as statistically significant. The only benchmark in which the specialization of objects produces a statistically significant speedup

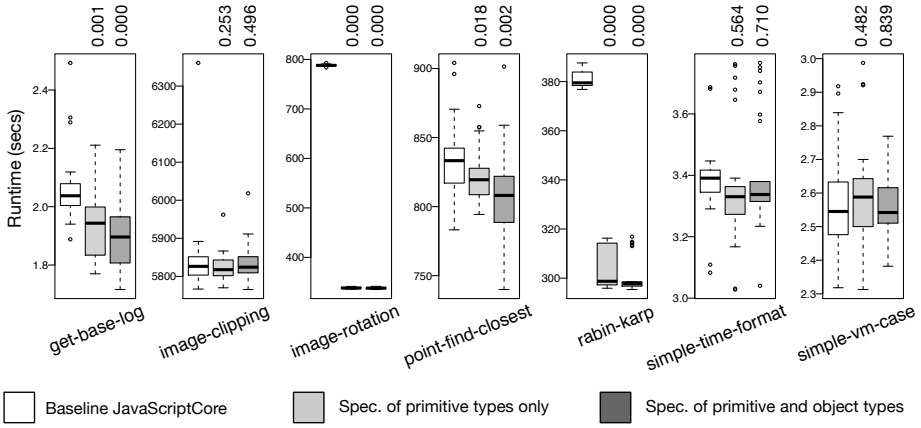

Figure 15: Absolute runtimes used to produce the ratios seen in Figure 14. Boxes for each benchmark represent, from left to right, the baseline implementation of JavaScriptCore, the optimized version with specialization of primitive types only, and the optimized version with object specialization. Numbers above boxes are p-values comparing the samples in that box against the samples in the baseline box.

over the specialization of primitives is `point-find-closest`[7]. In this case, the `_x` and `_y` properties of the query point can be specialized, and are never deoptimized henceforth. This speedup is not greater because JavaScriptCore is able to constant fold property accesses of objects once it reaches the FTL optimization level, if the property always returns the same value. In such scenario, both the baseline JIT and the object specializer generate almost the same code. The only difference is that object specialization can produce optimized code earlier.

On average (geo-mean), we got a speedup of 1.18x specializing primitive values and 1.20x when also specializing objects. If we remove `image-rotation`, clearly an outlier, then the speedups become 1.04x and 1.05x, respectively. This result lets us provide a positive answer to our first research question, e.g., guided

[7]To keep figures simple, we omit the p-values between populations from the two different optimizations. The version of `point-find-closest` optimized with object specialization was faster than the version that specializes only primitives, with a p-value of 0.009.

specialization can improve performance in some programs, even when applied onto a well-engineered compiler.

Function Memoization. The speedup of 2.4x observed in the benchmark image-rotation is due to constant folding. The core of this benchmark is a function rotate, which we show below (for simplicity, we remove some boilerplate code from this function):

```
function rotate(p, angle) {
    let newX = Math.cos(angle) * p.x - Math.sin(angle) * p.y;
    let newY = Math.sin(angle) * p.x + Math.cos(angle) * p.y;
    return {x: newX, y: newY};
}
```

Folding lets JavaScriptCore replace the calls to Math.sin and Math.cost with constants. JSC replaces calls to transcendental functions by constants whenever their actual parameters are constants, since these are pure, i.e,. side effect free, functions. To ensure correctness, JSC uses watchpoints to check for redefinitions of functions in the Math library. The combination of function memoization with value specialization is also responsible for some large speedups that we have observed in the JavaScriptCore Benchmark Collection. This optimization lets us, for instance, go beyond what can be accomplished with the polymorphic inline cache used by JavaScriptCore.

Runtime Variation in the JavaScriptCore Benchmark Collection. We have also observed a few large speedups in samples available in JavaScriptCore's benchmark collection. Our highest speedup in JBC was 2.25x on simple-getter-access; our heaviest slowdown was 23.90x on unprofiled-licm. This slowdown can be explained by the overhead introduced by the Specialization Cache. The cache triggers multiple compilations of a function and some microbenchmarks simply do not run long enough to pay this overhead off. As an example, unprofiled-licm, the benchmark that gave us the largest slowdown, runs for less than one millisecond.

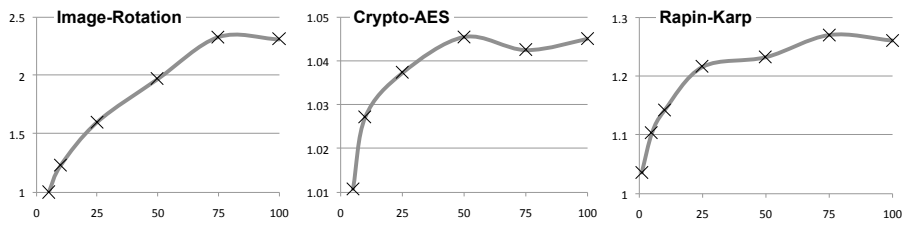

Figure 16: Variation of speedups with the number of invocations of the specialized function for three different benchmarks. The X-axis shows number of times each specialized function is invoked (1, 5, 10, 25, 50, 75 and 100 times). The Y-axis shows speedup, in number of times.

The Causes of Speedups. Specialization happened in every program where we have observed some speedup. Each dynamic instance of a function is specialized code that we produce. A new instance is produced for each different argument marked as invariant by the analysis of Section 3.1. We can create up to four new instances –the cache capacity, as described in Section 4.2.

We tend to obtain larger speedups in benchmarks that invoke the specialized function several times. As an example, Figure 16 shows how speedups vary with the number of invocations of the specialized function. We have manually changed each of these three benchmarks, so that the same specialized function could be called several times. Multiple invocations of the specialized function tend to amortize its compilation overhead, as Figure 16 indicates. Similar behavior is observed in specialized functions that run for a longer period of time. In other words, the hotter is the target function, the greater the speedup that we can achieve. A hot function is either a function that is called many times, or a function that contains one or more loops that run for a long time.

5.2. RQ2 – Size Analysis

To investigate the impact, in terms of space, of guided specialization, we have compared the memory footprint occupied by the native code that it generates, against the space occupied by the native code that the original version

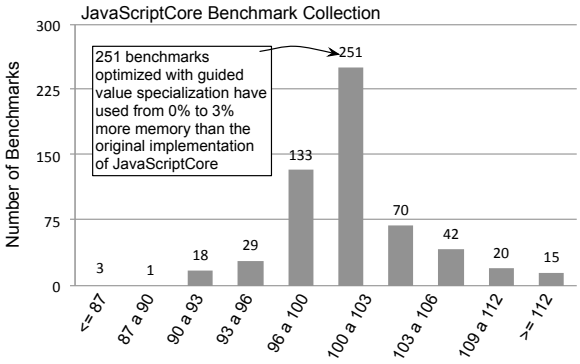

Figure 17: Guided specialization requires slightly more memory than the baseline JavaScript-Core. The graph shows a histogram of memory usage. X-axis is percentage of difference, in size, between baseline JavaScriptCore and our implementation, and Y-axis shows number of benchmarks.

of JavaScriptCore produces. The histogram in Figure 17 summarize our results. Figure 17 (b) for JavaScriptCore's benchmark collection.

Notice that we can, sometimes, produce smaller binaries. We have produced smaller binaries for 184 benchmarks, out of a universe of 582 programs in JBC. Larger binaries have been observed in 398 samples in the same collection. Size reduction happens as a consequence of the constant propagation coupled with dead-code elimination that our approach enables. Nevertheless, the contrary, i.e., code size expansion, is more common. Multiple specializations of the same function might lead to code duplication, although this phenomenon is rare among the benchmarks that we have. Therefore, answering our research question, guided specialization tends to increase the size of the binaries that the JIT compiler produces, although by a small margin.

5.3. RQ3 – Profitability Threshold

Our results might vary due to the profitability threshold that we adopt. We measure the profitability threshold in terms of CPU cycles. We approximate this metric by considering the static cost, in cycles, of every x86 instruction. Following the discussion in Section 3.2, this number of cycles gives us the pa-

rameter D in the profit of an instruction[8]. We assume that every loop runs 10 times, an approach that we acknowledge as overly simple, yet practical.

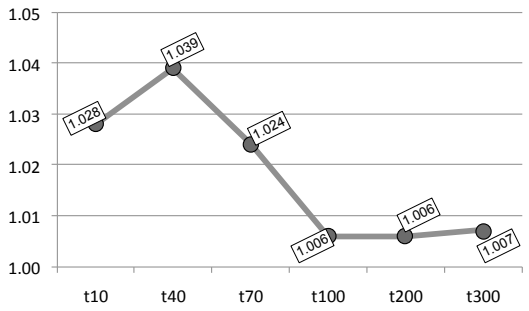

Figure 18: Influence of threshold on runtime. Numbers are speedup over baseline, in number of times.

Figure 18 gives us the result of this experiment, considering thresholds of 10, 40, 70, 100, 200 and 300 cycles, for the benchmarks in the JavaScriptCore Benchmark Collection. X-axis shows six different thresholds, and Y-axis shows the percentage of speedup observed for each threshold, counting only benchmarks that presented definite changes (speedups or regressions). The most effective threshold that we have observed across all the 582 benchmarks in JBC that we have executed was 40. Results also tend to vary per benchmark. This last observation lets us answer our research question: the choice of the profitability threshold is important to the quality of the code that we produce.

5.4. RQ4 – Allocation Stamps

The implementation of allocation stamps, as described in Section 4.1, imposes some overhead on the runtime environment that uses them. To find out how heavy this overhead is, we have disabled specialization, while keeping all the infrastructure necessary to carry it out. Figure 19 shows runtime results for the Synthetic benchmarks. Following JavaScriptCore's benchmarking methodology,

[8] D is part of Equation 1, in Section 3.2

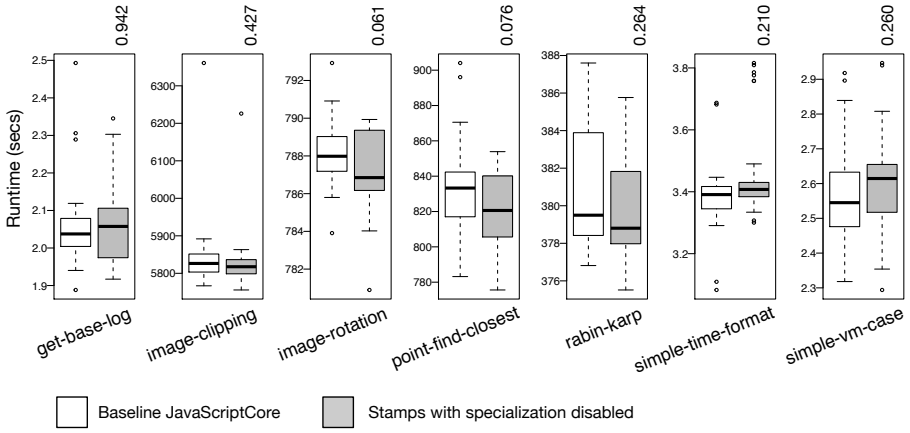

Figure 19: Overhead of allocation stamps. We are preserving all the infrastructure necessary to specialize objects, but we are not running any specialization. P-Values can be seen on the top of the boxplot for the transformed programs. None of these results can be considered statistically significant, because they give us a p-value above 0.01 –our significance threshold. Thus, speedups and slowdowns in this figure can be considered measurement noise.

no benchmark in Figure 19 was *definitely slower*.

Repeating this very experiment in JavaScriptCore's benchmark collection we observe a maximum overhead of 68% on "generator-with-several-types", a program that creates a list of objects of several different types. This program contains only a loop, in which objects are allocated on the heap, and represents the worst-case scenario to our implementation. In other words, several objects that would be specialized have been marked with allocation stamps; however, no specialization ever took place. On the positive side, we have not observed definite slowdowns in 559 benchmarks in JBC, out of 582. Thus, answering our research question, allocation stamps might bring a non-negligible overhead to particular benchmarks, but such situations are uncommon.

5.5. RQ5 – Unrestricted Specialization

Traditional value specialization [13] optimizes code blindly: functions are specialized for every argument. The values of the arguments are cached, and if

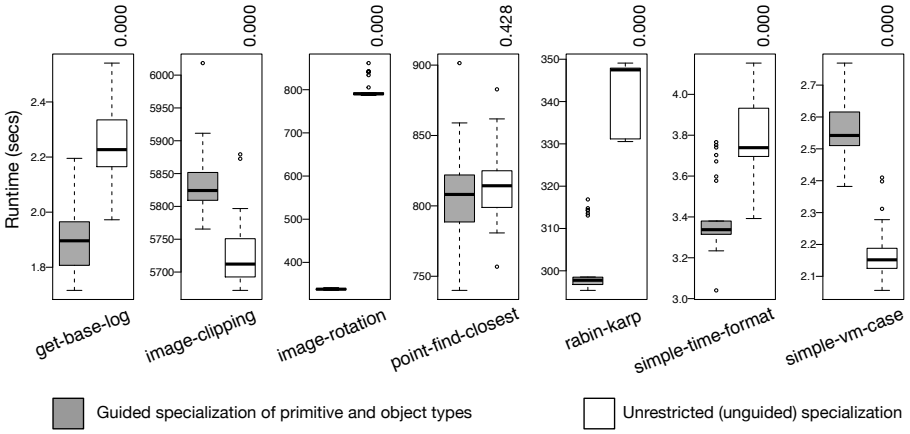

Figure 20: Guided specialization reduces the slowdowns experienced with unrestricted specialization. Y-axis shows runtime in seconds of programs compiled with guided and unrestricted value specialization. Guided specialization delivers faster code in 4 benchmarks, unrestricted does better in two of them, and one benchmark (`point-find-closest`) runs within the error margin (p-value above 0.05). Numbers above boxplots are p-values comparing both populations.

the function is called again, with the same values seen during specialization, the specialized version is invoked, otherwise, recompilation takes place (in case the generic binary has not been produced thus far). When we compare guided vs unrestricted specialization on Synthetic benchmarks, we observe runtime variation in several benchmarks. Figure 20 shows these results.

When we compare the two specialization strategies, we see also some large differences. Unrestricted specialization leads, in general, to worse results. However, there are situations in which unrestricted specialization does better, like in image-clipping. Situations like that happen due to the imprecision of the static analyses, which deem unprofitable code that would benefit from specialization. Nevertheless, Figure 20 indicates that guided specialization addresses the main shortcoming of its unrestricted counterpart: the inability to pinpoint code that benefits more from specialization. To give the reader an idea of this problem, in the 582 benchmarks available in JCB, we observed 228 deoptimizations (spe-

cialized code had to be recompiled) using unrestricted specialization, and only 12 using our guided methodology. As an example, in `image-rotation`, unrestricted specialization specializes the `main` routine, which, indeed, contains a hot function, but that, nevertheless, runs only a few times.

6. Related Work

Value Specialization at runtime. Ap *et al.* [15] presents a system that is able to apply value specialization dynamically onto C programs. To this end, Ap *et al.* attach a process to the running program. This process identifies hot functions. Once a hot function is identified, the monitor records the variance of the values that said function manipulates. When a specific value reaches a certain threshold, a version of the hot function is recompiled, and is specialized to that value. This approach uses a cache similar to the dynamic cache that we discussed in Section 4.2. Ap *et al.*'s technique is purely dynamic: targets of specialization are discovered at runtime. Our approach is hybrid: we use static analysis to identify such targets, while specialization happens, naturally, at runtime. A direct comparison between these two approaches is not trivial, as they run in completely different runtime systems. Nevertheless, as we shall mention in Section 7, we want to implement a purely dynamic approach in JavaScriptCore, to compare it against our hybrid technique as future work.

Using annotation to improve JIT compilation. Annotation systems have been used to aid the generation of specialized code before. In the late 90's, Grant *et al.* [35] proposed annotation-directed dynamic specialization for C. Their approach works by means of two pragmas that drive the run-time specialization: make_static and make_dynamic. The first specifies that some variables must be treated as run-time constants at all subsequent program points until reaching either another annotation or going out of scope. Azevedo *et al.* [36] presented an implementation of the Java Virtual Machine (JVM) that is annotation-aware. That annotation system would help a Java JIT compiler to improve the performance of register-allocation, common sub-expression elimination and value

propagation, by running these analyses offline and annotating the code to inform the compiler their results. An annotation-aware JIT compiler for Java is also the main focus in the work of Pominville *et al* [22]. Their approach is based on the insertion of code annotations in class file attributes present in the Java bytecode. As a result, the JIT module present in a JVM modified by them is able to eliminate array bound checks, sink objects, bind methods statically, and eliminate null pointer checks. Krintz *et al.* [37] use a different approach of annotation-oriented dynamic compilation to reduce the compilation time of Java functions. They define annotations to provide static information to the register allocator, to optimize memory usage and to indicate which methods should be inlined. In posterior work, Krintz *et al.* [38] use annotations to combine off-line and online profile information, to speed compilation up. None of this previous work uses annotations to guide code specialization; furthermore, none of them use static analysis to show annotations along the code yet to be JIT compiled. These two aspects of our work distinguish it from the related literature.

Partial Evaluation of dynamic languages. Partial evaluation [18] is a general methodology to specialize a program to part of its inputs. Our work is a form of partial evaluation. Nirkhe *et al.* [39] present a formal semantic of an imperative language designed for the domain of hard real-time systems and prove that the application of partial evaluation preserves this semantics. Shali *et al.* [40] have introduced a way to combine offline and online partial evaluation, using constant expressions defined by the programmer as seeds. These expressions make it possible to specialize a program to runtime values. Notice that while Shali *et al.*'s approach allows the annotation of any expression as constant, our approach annotates actual parameters only. Moreover, if our specialized code fails due to parameter changes, e.g. a specialized function is called again with different arguments, we deoptimize that function, whereas Shali *et al.* throw an error under such circumstances. Würthinger *el al.* [41, 42] also uses partial evaluation to carry out code optimizations. In their work, they present a new approach to build high performance Virtual Machine for high-level languages

where the programmer implements an interpreter of a given Abstract Syntax Tree (AST) using their framework. This framework, then enables JIT compilation of programs through partial evaluation of that interpreter. Finally, variance and profitability analyses are unique elements of this work.

7. Conclusion

This paper has presented guided just-in-time specialization, a technique to speed up the execution of code produced by a JIT compiler. Guided specialization brings forward two core ideas. First, it integrates static analysis and runtime support. This integration moves out of execution time the monitoring overhead necessary to identify specialization opportunities, while still ensuring the semantic correctness of specialized programs. Second, our guided approach goes beyond the more traditional unrestricted specialization model, producing specialized code only for function arguments that are likely to remain invariant.

An interesting question that this paper raises concerns how our ideas would fare, had we replaced our variance analysis with profiling. This comparison is not possible today in JavaScriptCore, because this runtime engine does not provide any profiling machinery that can be tailored to the needs of selective specialization of code. On the one hand, profiling tends to be more precise than static analysis; on the other, it misses a holistic view of the program and introduces runtime overhead. The advantages and disadvantages of each approach is a subject that we hope to investigate in the future.

Acknowledgement

This project was funded by grants from FAPEMIG (Grant APQ-03832-14 "Cooperation FAPs-INRIA-CNRS"), CNPq (Grant 406377/2018-9) and CAPES. We thank the referees for the time and expertise they have put into reviewing our work. Their suggestions have greatly improved our paper.

References

[1] B. Eich, Javascript at ten years, in: ICFP, ACM, New York, NY, USA, 2005, pp. 129–129.

[2] G. Richards, S. Lebresne, B. Burg, J. Vitek, An analysis of the dynamic behavior of javascript programs, in: PLDI, ACM, New York, NY, USA, 2010, pp. 1–12.

[3] R. Dewsbury, Google Web Toolkit Applications, Prentice Hall, Upper Saddle River, NJ, USA, 2007.

[4] S. Doeraene, T. Schlatter, N. Stucki, Semantics-driven interoperability between scala.js and javascript, in: SCALA, ACM, New York, NY, USA, 2016, pp. 85–94.

[5] A. Zakai, Emscripten: An llvm-to-javascript compiler, in: OOPSLA, ACM, New York, NY, USA, 2011, pp. 301–312.

[6] R. Auler, E. Borin, P. de Halleux, M. Moskal, N. Tillmann, Addressing javascript jit engines performance quirks: A crowdsourced adaptive compiler, in: CC, Springer, Heidelberg, Germany, 2014, pp. 218–237.

[7] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, M. Franz, Tracing for web 3.0: Trace compilation for the next generation web applications, in: VEE, ACM, New York, NY, USA, 2009, pp. 71–80.

[8] M. Chang, B. Mathiske, E. Smith, A. Chaudhuri, A. Gal, M. Bebenita, C. Wimmer, M. Franz, The impact of optional type information on JIT compilation of dynamically typed languages, SIGPLAN Not. 47 (2) (2011) 13–24.

[9] B. Hackett, S.-y. Guo, Fast and precise hybrid type inference for javascript, SIGPLAN Not. 47 (6) (2012) 239–250.

[10] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, O. Yamauchi, The hiphop virtual machine, in: OOPSLA, ACM, New York, NY, USA, 2014, pp. 777–790.

[11] A. Gal, B. Eich, M. Shaver, D. Anderson, B. Kaplan, G. Hoare, D. Mandelin, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmair, M. R. Haghighat, M. Bebenita, M. Change, M. Franz, Trace-based just-in-time type specialization for dynamic languages, in: PLDI, ACM, New York, NY, USA, 2009, pp. 465 – 478.

[12] M. R. S. Souza, C. Guillon, F. M. Q. Pereira, M. A. da Silva Bigonha, Dynamic elimination of overflow tests in a trace compiler, in: Compiler Construction, Springer, Heidelberg, Germany, 2011, pp. 2–21.

[13] I. R. de Assis Costa, P. R. O. Alves, H. N. Santos, F. M. Q. Pereira, Just-in-time value specialization, in: CGO, ACM, New York, NY, USA, 2013, pp. 1–11.

[14] U. Hölzle, C. Chambers, D. Ungar, Optimizing dynamically-typed object-oriented languages with polymorphic inline caches, in: ECOOP, Springer, Heidelberg, Germany, 1991, pp. 21–38.

[15] A. A. Ap, E. Rohou, Dynamic function specialization, in: SAMOS, ACM, Pythagorion, Samos, Greece, 2017, pp. 1–8.

[16] S. H. Jensen, A. Møller, P. Thiemann, Type analysis for javascript, in: SAS, Springer, Heidelberg, Germany, 2009, pp. 238–255.

[17] C. Humer, C. Wimmer, C. Wirth, A. Wöß, T. Würthinger, A domain-specific language for building self-optimizing ast interpreters, in: GPCE, ACM, New York, NY, USA, 2014, pp. 123–132.

[18] N. D. Jones, C. K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, 1st Edition, Prentice Hall, Upper Saddle River, NJ, USA, 1993.

[19] V. H. S. Campos, P. R. O. Alves, H. N. Santos, F. M. Q. Pereira, Restrictification of function arguments, in: CC, ACM, New York, NY, USA, 2016, pp. 163–173.

[20] G. Mendonça, B. Guimarães, P. Alves, M. Pereira, G. Araújo, F. M. Q. Pereira, Dawncc: Automatic annotation for data parallelism and offloading, ACM Trans. Archit. Code Optim. 14 (2) (2017) 13:1–13:25. doi:10.1145/3084540.

[21] P. Ramos, G. Souza, D. Soares, G. Araújo, F. M. Q. Pereira, Automatic annotation of tasks in structured code, in: PACT, ACM, New York, NY, USA, 2018, pp. 31:1–31:13. doi:10.1145/3243176.3243200.

[22] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, C. Verbrugge, A framework for optimizing java using attributes, in: CASCON, IBM Press, Armonk, NY, USA, 2000, pp. 8–24.

[23] D. E. Denning, P. J. Denning, Certification of programs for secure information flow, Commun. ACM 20 (1977) 504–513.

[24] A. W. Appel, J. Palsberg, Modern Compiler Implementation in Java, 2nd Edition, Cambridge University Press, Cambridge, UK, 2002.

[25] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, Boston, MA, US, 2006.

[26] L. George, A. W. Appel, Iterated register coalescing, TOPLAS 18 (3) (1996) 300–324.

[27] T. Ball, J. R. Larus, Branch prediction for free, in: PLDI, ACM, New York, NY, USA, 1993, pp. 300–313.

[28] Y. Wu, J. R. Larus, Static branch frequency and program profile analysis, in: MICRO, ACM, New York, NY, USA, 1994, pp. 1–11.

[29] M. Sagiv, T. Reps, S. Horwitz, Precise interprocedural dataflow analysis with applications to constant propagation, Theor. Comput. Sci. 167 (1-2) (1996) 131–170.

[30] S. Maffeis, J. C. Mitchell, A. Taly, An operational semantics for JavaScript, in: APLAS, Springer, Heidelberg, Germany, 2008, pp. 307–325.

[31] D. Park, A. Stefănescu, G. Roşu, KJS: A complete formal semantics of javascript, in: PLDI, ACM, New York, NY, USA, 2015, pp. 346–356.

[32] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the smalltalk-80 system, in: POPL, ACM, New York, NY, USA, 1984, pp. 297–302.

[33] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, TOPLAS 13 (4) (1991) 451–490.

[34] G. W. Hill, Algorithm 395: Students t-distribution, Commun. ACM 13 (10) (1970) 617–619.

[35] B. Grant, M. Mock, M. Philipose, C. Chambers, S. J. Eggers, Annotation-directed run-time specialization in c, in: PEPM, ACM, New York, NY, USA, 1997, pp. 163–178.

[36] A. Azevedo, A. Nicolau, J. Hummel, Java annotation-aware just-in-time (ajit) complilation system, in: Java Grande, ACM, New York, NY, USA, 1999, pp. 142–151.

[37] C. Krintz, B. Calder, Using annotations to reduce dynamic optimization time, in: PLDI, ACM, New York, NY, USA, 2001, pp. 156–167.

[38] C. Krintz, Coupling on-line and off-line profile information to improve program performance, in: CGO, IEEE, Washington, DC, USA, 2003, pp. 69–78.

[39] V. Nirkhe, W. Pugh, Partial evaluation of high-level imperative programming languages with applications in hard real-time systems, in: POPL, ACM, New York, NY, USA, 1992, pp. 269–280.

[40] A. Shali, W. R. Cook, Hybrid partial evaluation, in: OOPSLA, ACM, New York, NY, USA, 2011, pp. 375–390.

[41] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, M. Wolczko, One VM to rule them all, in: Onward!, ACM, New York, NY, USA, 2013, pp. 187–204.

[42] T. Würthinger, C. Wimmer, C. Humer, A. Wöß, L. Stadler, C. Seaton, G. Duboscq, D. Simon, M. Grimmer, Practical partial evaluation for high-performance dynamic language runtimes, SIGPLAN Not. 52 (6) (2017) 662–676.