Interprocedural Type Specialization of JavaScript Programs Without Type Analysis*

Maxime Chevalier-Boisvert[1] and Marc Feeley[2]

1 **DIRO, Université de Montréal**
Montreal, Quebec, Canada
chevalma@iro.umontreal.ca
2 **DIRO, Université de Montréal**
Montreal, Quebec, Canada
feeley@iro.umontreal.ca

Abstract

Previous work proposed lazy basic block versioning, a technique for just-in-time compilation of dynamic languages which we believe represents an interesting point in the design space. Basic block versioning is simple to implement, simple enough that a single developer can build a complete just-in-time compiler for JavaScript in a year, yet it performs surprisingly well as it propagates context-sensitive type information to generate type-specialized code on the fly.

In this paper, we demonstrate that lazy basic block versioning can be extended in simple ways to propagate type information across function call boundaries. This gives some of the benefits of whole-program analysis, or a tracing compiler, without having to implement the machinery for either. We have implemented this proposal in the Higgs JavaScript virtual machine and report on the empirical evaluation of this system on a set of industry standard benchmarks. The approach eliminates 94.3% of dynamic type tests on average, which we show is more than what is achievable with any static whole-program type analysis.

1998 ACM Subject Classification D.3.4 Programming Languages: Processors—compilers, optimization, code generation, run-time environments

Keywords and phrases Just-In-Time Compilation, Dynamic Language, Optimization, Object Oriented, JavaScript

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.7

1 Introduction

A production compiler for a widely used dynamic language such as JavaScript is an intricate piece of software, usually the outcome of 10 to 100 developer-years of effort. The architecture of such a compiler is one of the first design decisions made during development. This decision is rarely revisited, as architectural changes tend to be disruptive. In previous work, Chevalier-Boisvert and Feeley argued for an architecture based on the concept of lazy Basic Block Versioning (BBV) [14]. They claimed that the technique hits a sweet spot in the tradeoff between implementation complexity and performance of the generated code. As evidence they designed and implemented Higgs, a JavaScript virtual machine and Just-In-Time (JIT) compiler which has performance competitive with other research virtual machines and can sometimes match the performance of production systems such as V8. Notably, the

* This work was supported, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Mozilla Corporation.

30th European Conference on Object-Oriented Programming (ECOOP 2016).
Editors: Shriram Krishnamurthi and Benjamin S. Lerner; Article No. 7; pp. 7:1–7:24

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Higgs compiler took about a year of development time. The reduced development time is particularly important for languages that are maintained by small teams of volunteers. Lazy BBV occupies a point in the design space of JIT compilers that is between method-based compilers and tracing JITs such as Mozilla's TraceMonkey [17], and run-time specialization of Oracle's Truffle [36]. The simplicity of BBV is one of its main advantages. It does not require additional infrastructure such as a static analyzer to approximate program facts, or an interpreter to record traces.

BBV is a simple and elegant compilation technique to optimize dynamically typed programs on the fly. The technique uses dynamic type tests which are part of the implicit semantics of primitive operators in dynamically typed languages to capture and propagate type information. Type-specialized versions of individual basic blocks are lazily compiled based on the types encountered during the execution of programs. The technique, as described in [14], is limited to optimizing type checks on local variables within a single function. The compiler has no information on the types of arguments, return values, or object properties, and is thus unable to eliminate some redundant dynamic type checks.

This paper extends basic block versioning with the ability to propagate type information across function call boundaries and to specialize code based on the type of object properties. In the framework of basic block versioning, these extensions are easy to implement and seem to work rather well. This paper makes the following specific contributions:

1. The combination of BBV with a *typed object shape* mechanism which encodes property type information including *method identity*, enabling the compiler to know the identity of callees at call sites (Section 4.1).
2. The extension of BBV with *specialized function entry points*, which makes it possible to pass argument types from callers to callees. This is done efficiently, without dynamic dispatch, using method identity information provided by typed object shapes (Section 4.2).
3. A speculative technique for *call continuation specialization*, which enables type information about return values to be passed from callees back to callers, without dynamic overhead (Section 4.3).

To validate our claims we implemented these contributions in the Higgs JavaScript compiler and evaluated its performance on industry standard benchmarks (Section 5).

A word about evaluation is in order. We considered implementing our ideas within an existing JavaScript compiler, but quickly realized that the architectural changes required were beyond our resources. Thus we picked Higgs as a vehicle for our experiments. This choice comes at a cost; comparing performance of a research prototype to a production system is tricky. A production system has a mature garbage collector, highly tuned libraries, and performs a massive number of optimizations (many, but not all, of which are orthogonal to this work). A research prototype is likely to not have any of those. It is thus not surprising that Higgs runs roughly half as fast as V8. This may be a sign that our approach is inherently limited, or that we simply lack the resources of major corporations. Cognizant of the inherent limitations of empirical evaluations, we have chosen the following approach. We measure the improvement of the techniques presented in this paper by the number of type tests we are able to eliminate and the performance impact over the previous version of the Higgs compiler. This gives us a metric of progress. We compare our implementation with two relevant systems, one is the TraceMonkey tracing compiler. The reason for this comparison is that basic block versioning has been compared by others to tracing compilation. It is thus interesting to see how the two perform on the same benchmarks. Then we choose Truffle/JS as an example of a research prototype, albeit one implemented by a large team of industrial

researchers. For completeness we include, in Appendix A, performance results comparing Higgs to leading commercial JavaScript implementations.

2 Influences and Related Work

The literature on just-in-time compilation is rich with, by now, decades of work. The work presented here was influenced by many results obtained in the Self project and should be contrasted to work on type analysis and dynamic compilation of dynamic languages.

Shapes. The notion of describing objects with shapes can be traced back to the Self programming language [11, 22], where so-called maps group objects cloned from the same prototype. Like shapes, maps reduce memory usage and stored metadata relating to properties (though not type information). Today, commercial JavaScript implementations such as V8, SpiderMonkey, Nitro and Truffle/JS have all adopted this idea. Each object contains a pointer to its shape, which describes the layout of the object and property attribute metadata. Truffle introduced the notion of specializing shapes based on property types to the literature [35]. This paper builds on that idea and demonstrates how to effectively integrate such a model with basic block versioning.

Splitting. Basic block versioning bears resemblance to Self's *iterative type analysis* and *extended message splitting* [13] which combines static analysis with a transformation that compiles multiple versions of loops and duplicates control flow paths to eliminate type tests. The analysis works in an iterative fashion, transforming the control flow graph of a function while performing a type analysis. It integrates a mechanism to generate new versions of loops when needed, and a message splitting algorithm to try and minimize type information lost through control flow merges. One key disadvantage is that statically cloning code requires being conservative, generating potentially more code than necessary, as it is impossible to statically determine exactly which control flow paths will be taken at run time, and this must be overapproximated. The approach also has roots in Agesen's cartesian product algorithm [2] which avoids the loss of type information at control-flow merges by representing program state with sets of vectors of concrete types.

Analysis. There have been multiple efforts to devise type analyses for dynamic languages. Rapid Atomic Type Analysis [27] is an intraprocedural flow-sensitive analysis that assigns unique types to each variable. Attempts have also been made to define formal semantics for a subset of dynamic languages such as JavaScript [5], Ruby [16] and Python [4], sidestepping some of the complexity of these languages and making them more amenable to traditional type inference techniques. There are also flow-based interprocedural type analyses for JavaScript based on sophisticated type lattices [23, 24, 25]. Such analyses are too time consuming to be used in a just-in-time compiler. Kedlaya, Roesch et al. [26] improved the precision of type analyses by combining them with type feedback and profiling. This shows promise, but does not deal with object shapes and property types. Work has also been done on a flow-sensitive alias analysis for dynamic languages [19], but it is still unclear if the analysis can be used on-line. More recently, Brian Hackett et al. presented an interprocedural hybrid type analysis for JavaScript suitable for use in a just-in-time compiler [21]. While this is an important step forward, it remains vulnerable to imprecise type information polluting analysis results. Basic block versioning can help improve on the results of such an analysis by hoisting tests out of loops and generating multiple optimized code paths where appropriate.

Tracing. Trace compilation, introduced by Dynamo [6] and later applied to just-in-time compilation in HotpathVM [18], aims to record sequences of instructions executed inside hot loops. Such sequences make optimization simpler. Type information is accumulated along traces and used to specialize code and remove type tests [17], overflow checks [34] or unnecessary allocations [8]. Basic block versioning resembles tracing in that context updating works on essentially linear code fragments and code is optimized similarly to what may be done in a tracing compiler. Code is also compiled lazily, as needed, without compiling whole functions at once. Trace compilation [9] and meta-tracing are an active area of research [10]. The simplicity of basic block versioning is one of its main advantages. It does not require external infrastructure such as an interpreter to record traces. Trace compiler implementations must deal with corner cases that do not appear with basic block versioning. With trace compilation, there is the potential for trace explosion if there are a large number of control flow paths going through a loop [7]. It is also not obvious how many times a loop should be recorded or unrolled to maximize the elimination of type checks. This problem is solved with basic block versioning since versioning is driven by type information and there is a natural bound to the number of versions that comes from the finite number of types in the system. Trace compilers must implement parameterizable policies and mechanisms to deal with recursion, nested loops and potentially very long traces that do not fit in instruction caches.

Customization. Customization is another technique developed to optimize Self programs [12]. It compiles multiple copies of methods specialized on the receiver object type. Similarly, *type-directed cloning* [28] clones methods based on argument types, producing more specialized code using richer type information. The work of Chevalier-Boisvert et al. on just-in-time specialization for MATLAB [15] and similar work done for the MaJIC MATLAB compiler [3] tries to capture argument types to dynamically compile optimized versions of whole functions. All of these techniques are forms of type-driven code duplication aimed at extracting type information. Basic block versioning operates at a lower level of granularity, allowing it to find optimization opportunities inside of method bodies by duplicating code paths. There are similarities between the Psyco JIT specialization work and our own. The Psyco prototype for Python [31] interleaves execution and JIT compilation to gather run time information about values. It then specializes code on the fly based on types and values. It also incorporates a scheme where functions can have multiple entry points. We extend upon this work by combining a similar approach, that of basic block versioning, with typed shapes and a mechanism for propagating return types from callees to callers with low overhead. The *tracelet-based* approach used by Facebook's *HHVM* for PHP [1] bears similarities to our own. HHVM compiles small code regions (tracelets) which are single-entry multiple-exit basic blocks. Each tracelet is type-specialized based on variable types observed at compilation time. Guards are inserted at the entry of tracelets to verify at run time that the types observed are still valid for all future executions. High-level instructions in tracelets are specialized based on the guarded types. If these guards fail, new versions of tracelets are compiled based on different type information and chained to the failing guards. One difference with our work is that HHVM uses an ahead-of-time type analysis pass. Another difference is that with the approach described in [1], each tracelet re-checks the types of its inputs, whereas BBV propagates known types to successor blocks and doesn't usually need to re-check the types of local variables. Finally, HHVM falls back on an interpreter when too many tracelet versions are generated. Higgs falls back to generic basic block versions which do not make type assumptions but are still compiled. Beyond type specialization, recent work by Costa et al. on *just-in-time value specialization* has shown that specializing JavaScript functions based

on specific argument values can lead to performance improvements [33], as many functions are always called with the same arguments.

3 Background

The work presented in this paper is implemented in a research virtual machine for JavaScript (ECMAScript 5) known as Higgs [1]. The Higgs virtual machine includes a just-in-time compiler built around lazy basic block versioning. This compiler is intended to be lightweight with a simple implementation. Code generation and type specialization are performed in a single pass. Register allocation is done using a greedy allocator. The runtime and standard libraries are self-hosted, written in an extended dialect of JavaScript with low-level primitives. These low-level primitives are special instructions which allow expressing type tests, pointer manipulation, as well as integer and floating point machine instructions in the source language.

3.1 Value Types and Type Tests

Higgs segregates values into categories based on type tags [20]. These type tags form a simple type system that is used for versioning. The types are mostly straightforward and correspond closely to values manipulated by JavaScript programs. The one exception is the `unknown` type tag that is used by the compiler to indicate that no information is available for the corresponding value.

`int32`	signed 32-bit integers
`float64`	64-bit floating point numbers
`undef`	the `undefined` value
`null`	the `null` value
`bool`	`true` and `false` boolean values
`string`	strings
`array`	arrays
`closure`	function objects
`object`	Plain JS objects
`unknown`	type unknown

JS is a dynamically typed and late-bound programming language. There are no static type annotations, and the types of variables may change during the execution of a program. As such, there are many implicit type checks hiding in even the simplest JS programs. Figure 1 shows an iterative function which illustrates this. The `sum` function contains three primitive operators: a comparison, a decrementation and an addition. Each of these operators implicitly checks the types of its operands as part of its semantics.

In all, there are four implicit type checks hiding in the `sum` function:

1. The `>` operator checks the type of `n` before comparing it against the integer zero.
2. The type of `s` is checked before computing `s += i`
3. The type of `i` is also checked before computing `s += i`
4. The decrementation operator checks the type of `i` before computing `--i`

[1] https://github.com/higgsjs

```
function sum(n) {
    var s = 0;
    for (var i = n; i > 0; --i)
        s += i;
    return s;
}

sum(500);
```

Figure 1 Iterative JS `sum` function.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I  // if not (n > 0)

C: if not is_int32(s) goto stub2
D: if not is_int32(i) goto stub3
E: s = add_int32(s,i)
   if overflow goto stub4

F: if not is_int32(i) goto stub5
G: i = sub_int32(i,1)
   if overflow goto stub6
H: goto B

I: return s
```

Figure 2 Control-flow graph of the `sum` function before BBV.

A naive JS implementation performs these type checks every time an operator is evaluated. In Higgs, this is done using primitive instructions which can test the type tags of values. Figure 2 illustrates the primitive operations and implicit type tag checks executed by Higgs with basic block versioning disabled when `sum(500)` is evaluated. When computing `sum(500)`, only small integer (`int32`) values are used, and so, much of these type checks are redundant.

The `is_int32` primitives act as guards which verify that the type tag associated with a given variable is `int32` before executing a machine instruction specific to integer values. Should any of these tests fail, execution will flow to a stub that generates new machine code to handle non-integer values. The overflow test primitives serve to verify that an integer overflow did not occur, and handle such an occurrence otherwise.

3.2 Lazy Basic Block Versioning

Basic block versioning is a just-in-time code generation technique originally applied to JavaScript by Chevalier-Boisvert & Feeley [14], and adapted to Scheme by Saleil & Feeley [32]. The technique bears similarities to HHVM's tracelet-based compilation approach and Psyco's just-in-time code specialization system [31].

BBV works at the level of individual basic blocks. We define a basic block as a single-entry single-exit sequence of instructions. Basic blocks end with one branching instruction which jumps to other basic blocks. In Higgs, basic blocks are usually short, sometimes just one instruction in our Intermediate Representation (IR), due to the large number of type tests, each of which is treated as a branching instruction which terminates the current basic block.

The BBV engine interleaves compilation and execution. It generates machine code for basic blocks lazily, instantiating them into one or more versions, each type-specialized based on accumulated type information. BBV propagates type information by maintaining a context for each block version which stores known type information about live variables.

```
A: s = 0, i = 0
   if not is_int32(n) goto stub1
B: if not gt_int32(n,0) goto I

// s,i,n are known to be int32
C: //is_int32(s) check eliminated
D: //is_int32(i) check eliminated
E: s = add_int32(s,i)
   if overflow goto stub2

// s,i,n are known to be int32
F: //is_int32(i) check eliminated
G: i = sub_int32(i,1)
   if overflow goto stub3
H: goto B

I: return s
```

Figure 3 Control-flow graph of the `sum` function after BBV.

This context is updated as block versions are compiled.

Type tag tests are used to capture type information and enrich the versioning context. We know, for instance, that if we branch on the "true" side of an `is_int32(n)` test, then `n` must have tag `int32` in the successor block. This fact is exploited by instantiating a specialized version of the successor block based on the knowledge that `n` is `int32`. Because BBV uses lazy code generation, it never generates block versions for types that do not occur at run time. It achieves this by delaying the compilation of conditional branch targets using machine code stubs.

Using BBV, three of the four implicit type checks in the `sum` function from Figure 1 are eliminated. The resulting optimized control flow graph is shown in Figure 3. A single type test remains: the type of `n` is tested when entering the function. When first executing the `sum(500)` call, Higgs takes the following steps to compile and optimize the `sum` function:

- The `sum` function is entered, block `A` is executed. The `s` and `i` variables are initialized to `0`. The context is updated to indicate both `s` and `i` have type tag `int32`. The type of `n` is unknown. The `is_int32(n)` branch is made to point to machine code stubs and execution is resumed.
- Execution resumes. The `is_int32(n)` check evaluates to "true". A stub for block `B` is hit. This stub calls back into the compiler.
- Compilation resumes, and a version of block `B` with `n` known to be `int32` is generated. Stubs are generated for the `gt_int32(n,0)` branch targets.
- Execution resumes. A stub of block `C` is hit.
- Compilation resumes. A version of block `C` with `n` known to be `int32` is produced. The variables `s` and `i` are already known to be `int32`, hence the type tag checks in `C` and `D` can be evaluated at compilation time and eliminated. A stub is produced for the integer overflow check.
- Execution resumes. No overflow occurs, a stub for block `F` is hit.
- Compilation resumes. A version of block `F` with `s`, `i` and `n` as `int32` is compiled. The type check in `F` is evaluated at compilation time and eliminated. Stubs for the overflow branch in `G` are produced.
- Execution resumes. No overflow occurs, a stub of block `H` is hit.
- Compilation resumes. Block `H` produces a jump to the version of `B` that was already generated, where `s`, `i` and `n` are all known to be `int32`.

```
function sumList(lst) {
    if (lst == null)
        return 0
    return lst.val + sumList(lst.next)
}

function makeList(len) {
    if (len == 0)
        return null
    return { val: len, next: makeList(len-1) }
}

var lst = makeList(100)
if (sumList(lst) != 5050)
    throw Error('incorrect␣sum')
```

Figure 4 JS function to recursively sum the values stored in a linked list.

- Execution resumes and continues until the `gt_int32(n,0)` test in block `B` fails. Note that no more type checks are executed.
- The loop test fails. A stub for block I is hit. Block `I` is compiled.
- Execution resumes at block `I`, the `sum` function returns to the caller.

Because of its JIT nature, BBV has at least two powerful advantages over traditional static type analyses. The first is that BBV considers only the parts of the control flow graph that get executed, and it knows precisely which they are, as machine code is only generated for basic blocks which are executed. The second is that code paths can often be duplicated and specialized based on different type combinations, making it possible to avoid the loss of precision caused by control flow merges in traditional type analyses.

3.3 Motivating Example

The example in Figure 1 is one for which plain intraprocedural BBV works particularly well. In this section, we will provide a motivating example for our work which highlights the limitations of the unextended BBV approach described in [14]. We will then show how we have extended BBV to remove these limitations.

Figure 4 shows the `sumList` function for recursively traversing a linked list and computing the sum of numerical values stored in each node. While this small program may appear simplistic, there is much semantic complexity hidden behind the scenes. A correct but naive implementation of this function contains six implicit dynamic type tag tests, which must be eliminated to maximize performance:

1. The tag of the `lst` argument is checked when comparing it against `null`.
2. The tag of `lst` is re-checked before reading the `lst.val` property.
3. The tag of `lst` is checked a third time before reading the `lst.next` property.
4. The `sumList` function is a mutable global variable. Before calling it, there is an implicit check to make sure that this is in fact a closure.
5. The tag of `lst.val` is checked before computing `lst.val + sumList(lst.next)`.
6. The tag of `sumList(lst.next)` is also checked, because functions calls can return values of any type.

The BBV algorithm described in [14] is limited to an intraprocedural scope, that is, it deals with local variable types only. It cannot pass type information between callers and callees. It also assumes that all object properties (including global variables, which are

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: val = read_prop(lst, 'val')
   if not is_object(lst) goto stub2

D: next = read_prop(lst, 'next')
   sumfn = read_prop(globalObj, 'sumList')
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if overflow goto stub6

H: return t2

I: return 0
```

Figure 5 Implicit type checks in the `sumList` function.

properties of the global object) have unknown type. As such, the unextended BBV algorithm is ill-equipped to optimize the `sumList` function, or object-oriented JS code in general.

The implicit tests executed by a version of Higgs without BBV are shown in Figure 5.

Once the type tag of the `lst` parameter has been tested and found to be `object`, intraprocedural BBV can eliminate the second `is_object` test. Unfortunately, it cannot eliminate any of the other type tag tests. Since nothing is known about object property types, the type tags of the `val` and `next` properties must be tested for each call. The type tag of `sumList` is also tested before every call. Lastly, the return type of the `sumList` call is checked after each call. Clearly, most of these checks are provably redundant, and it should be feasible to eliminate them. The next sections will explain the ways in which we have extended BBV to give it the necessary capabilities.

4 Interprocedural Basic Block Versioning

This section describes the three extensions to basic block versioning that allow us to propagate type information across procedure calls.

4.1 Typed Object Shapes

BBV, as presented in [14], deals with function parameter and local variable types only. It has no mechanism for attaching types to object properties. This is particularly problematic because, in JS, functions are typically stored in objects. This includes object methods and also global functions (JS stores global functions as properties of the *global object*). We would like to attach type tags to object properties, global variables included.

4.1.1 Object Shapes and Shape Tests

Currently, all commercial JS engines have a notion of object shapes, which is similar to the notion of property maps invented for the Self VM. That is, any given object contains a pointer to a shape descriptor providing its memory layout: the properties it contains, the

```
// Linked list node shape
S: { val: slot 0, next: slot 1 }

// Global object shape
G: {
    ...,
    Error: slot 1,
    ...,
    makeList: slot 30,
    sumList: slot 33,
    lst: slot 34
}
```

Figure 6 Linked list node and global object shapes.

```
A: if is_null(lst) goto I

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S) call updatePIC  // PIC 1
   val = read_slot(lst, 0)                 // PIC 1
   if not is_object(lst) goto stub2

D: if not is_shape(lst, S) call updatePIC  // PIC 2
   next = read_slot(lst, 1)                // PIC 2
   if not is_shape(globalObj, G) call updatePIC // PIC 3
   sumfn = read_slot(globalObj, 33)            // PIC 3
   if not is_closure(sumfn) goto stub3

E: t1 = sumfn(next)
   if not is_int32(val) goto stub4

F: if not is_int32(t1) goto stub5

G: t2 = add_int32(val, t1)
   if not overflow goto stub6

H: return t2

I: return 0
```

Figure 7 Primitive operations in `sumList` executed by an unextended version of Higgs.

property slot index (memory offset) each property is stored at, as well as attribute flags (i.e. writable, enumerable, etc.). For instance, linked list nodes and the global object in the example from Figure 4 have shapes `S` and `G`, shown in Figure 6.

Traversing shape data structures on each object property access would be prohibitively expensive. As such, Higgs and all modern JS engines optimize property accesses using Polymorphic Inline Caches (PICs) [22]. PICs are lazily updated sequences of inlined machine instructions which implement property reads and writes. Typically, a cascade of conditional branch instructions establish the shape of an object in order to determine the memory offset at which the property to be read or written is stored. A specialized machine instruction is then executed which accesses the property at the correct offset. PICs are extended as needed to handle previously unseen object shapes.

In the `sumList` function, there are three property reads, and therefore three PICs. Linked list nodes and the global object only have one possible shape, and so there is only one shape test inside each PIC. The primitive operations and dynamic tests executed by an unextended implementation of Higgs which uses PICs are illustrated in Figure 7.

```
// Linked list node shape
S1: { val: (slot 0, int32), next: (slot 1, null) }
S2: { val: (slot 0, int32), next: (slot 1, object) }

// Global object shape
// Closures have method identity information
G: {
    ...,
    Error: (slot 1, closure/Error),
    ...,
    makeList: (slot 30, closure/makeList),
    sumList: (slot 33, closure/sumList),
    lst: (slot 34, object)
}
```

Figure 8 Typed object shapes encode property type information.

4.1.2 Extending Shapes with Types

Work done on the Truffle Object Model (OSM) [35] describes how object shapes can be straightforwardly extended to also encode type tags for object properties. Property writes are guarded to update object shapes when a property type changes. Property reads establish the shape of objects in order to know the memory offset at which to read properties. When object shapes also encode the type tags of properties, establishing the shape of an object tells us not only where to read the property, but also what type tag this property has. Hence, the cost of guarding property writes is easily offset, because typical JS programs have many more property reads than property writes. A small overhead is paid to guard property writes, and in exchange, type checks after property reads are effectively eliminated.

We extend upon the original BBV work with a *typed object shape* system inspired by the Truffle OSM. This model is a natural fit for the BBV algorithm. Our extended BBV algorithm not only propagates known type tags associated with values, but also object shapes. The shape tests which are normally part of PICs allow our JIT compiler to establish and propagate the shape of an object in the same way that type tag tests enabled BBV to extract and propagate the type tags of values. Once the shape associated with an object is known to the BBV engine, then, by extension, the types of all properties read from that object are also known.

In order to enable interprocedural type propagation, it is useful to know which function is being called for as many call sites as possible, both for calls to global functions and method calls. As such, we have gone one step further than the Truffle OSM, and attached not only type tags to object shapes, but also *method identity information*. That is, for properties which have the closure type tag, shapes encode a pointer to the IR node corresponding to the function the property is a closure of. This enables us to know the identity of callees at code generation time for the large majority of call sites.

With typed shapes, linked list nodes from the `sumList` have two possible shapes, one where the `next` property is `null`, and one where it is an object. The global object encodes not only the offsets of global variables, but also the identity of global functions. This is illustrated in Figure 8.

In order to allow BBV to take advantage of typed shape information, we break up PICs into their component parts. PICs, which were previously monolithic sequences of inlined machine instructions, are now exposed in our compiler IR as separate shape test and memory access instructions. The result is that the regular BBV mechanisms can be leveraged to extract shape information from shape tests and propagate it. Propagating shape information (and the associated property types), allows us to optimize the `sumList` function as shown in Figure 9.

```
A: if is_null(lst) goto I:

B: if not is_object(lst) goto stub1
C: if not is_shape(lst, S1) goto C2
   val = read_slot(lst, 0) // val is known to be int32
   next = read_slot(lst, 1) // next is known to be null

D: if not is_shape(globalObj, G) goto stub2
   sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E: t1 = sumfn(next)
   if not is_int32(t1) goto stub3

G: t2 = add_int32(val, t1)
   if overflow goto stub4

H: return t2

I: return 0

C2: if not is_shape(lst, S2) goto stub5
    val = read_slot(lst, 0) // val is known to be int32
    next = read_slot(lst, 1) // next is known to be object

D2: if not is_shape(globalObj, G) goto stub6
    sumfn = read_slot(globalObj, 33) // sumfn is known to be a closure

E2: t1 = sumfn(next)
    if not is_int32(t1) goto stub7

G2: t2 = add_int32(val, t1)
    if overflow goto stub8

H2: return t2
```

Figure 9 The `sumList` function optimized with typed shapes.

Two separate code paths are generated inside the `sumList` function, one for each of the two possible shapes of the linked list nodes. More code is generated, but on any given code path, at most three type tag tests are executed instead of five. Since linked list nodes now have two possible shapes, we may test the shape of linked list nodes twice instead of just once when reading the `lst.val` property. However, because we no longer employ monolithic inline caches, this shape is propagated from the property read of `lst.val` to that of `lst.next`. Hence, as a result, we actually perform less dynamic shape tests on average.

4.2 Entry Point Versioning

Procedure cloning has been shown to be a viable optimization technique, both in ahead of time and JIT compilation contexts. By specializing function bodies based on argument types at call sites, it becomes possible to infer the types of a large proportion of local variables, allowing effective elimination of type checks.

Our first extension to BBV is to allow functions to have multiple type-specialized entry points. That is, when the identity of a callee at a given call site is known at compilation time, the JIT compiler requests a specialized version of the entry point block for the callee. This specialized entry point assumes the argument types known at the call site. Type information is thus propagated from the caller to the callee.

Inside the callee, BBV proceeds as described in [14], deducing local variable types and eliminating redundant type checks. Our approach places a hard limit on the number of

versions that may be created for a given basic block, and so automatically limits the number of entry points that may be created for any given function. If there are already too many specialized entry points for a given callee, a generic entry point is obtained instead. This does not matter to the caller and occurs rarely in practice.

Propagating types from callers to callees allows eliminating redundant type tests in the callee, but also makes it possible to pass arguments without boxing them, thereby reducing the overhead incurred by function calls. Note that our approach does not use any dynamic dispatch to propagate type information from callers to callees. It relies on information obtained from typed shapes to give us the identity of callees (both global functions and object methods) for free. When the identity of a callee is unknown, a generic entry point is used.

In the case of the linked list example from Section 3.3, we can specialize the `sumList` function entry point based on the type tag of the `lst` parameter. As a consequence, we know whether `lst` has tag `null` or `object` upon entering the function.

With entry point versioning, we can eliminate all type tag checks, except for the check on the return type of the `sumList` call. This test seems redundant, considering that, in our example, the `sumList` function only ever returns `int32` values. The following section will explain our strategy to optimize this.

4.3 Call Continuation Specialization

Achieving full interprocedural type propagation demands passing the return type information from callees to callers. While it is fairly straightforward to establish the identity of the callee a call site will jump to in the majority of cases, establishing where a `return` statement will jump to is less straightforward. This is to say, most call sites are monomorphic and jump to a single function, and hence, a single specialized entry point. Furthermore, versioning code based on object shapes has the net effect that it will often split polymorphic call sites into monomorphic ones, which is very convenient for us.

We would like to version call continuations (the code executed when we return from a call) in accorance with the return types observed during execution. However, one `return` statement can potentially jump to several call continuations within a program. This means we cannot employ the same strategy as with entry point versioning. We cannot simply jump from one `return` statement to a specialized call continuation which assumes a known return type. Type information about return values could be propagated with a dynamic dispatch of the return address indexed with the result type. However this would incur a run time cost. We would be trading one form of dynamic overhead (that of type checks) for another (that of dynamic dispatch).

Instead, we have chosen to extend BBV with an approach that has zero run time cost (amortized overhead). Call continuations are compiled lazily when the first return to a given continuation is executed. When a function first executes a `return` statement, its return type, if known, is memorized. Call continuations are then speculatively optimized based on this memorized return type. If later returns from this function turn out to have a different return type, the optimized call continuations are invalidated (see Section 4.3.1).

Given the small example given in Figure 10 where a function `f` calls some function `g`, where `g` always returns values of type `int32`, the call continuation specialization process continuations takes the following steps:

- A call to `g` is encountered. Assuming the identity of the callee is known from typed shapes (otherwise this optimization is not performed), `f` is added to a list of callers of `g`.

```
function f() {
    // Call site
    var r = g()

    // Call continuation
    // The addition has an implicit type check
    return r + 2
}

function g() {
    return 1
}
```

Figure 10 Function call with a fixed return type.

- A stub is generated for the call continuation in `f`.
- Machine code for the call site is generated, it is made to jump directly to a specialized entry point in `g`.
- Execution resumes in `f` and jumps to `g`. Execution continues until `g` returns.
- Compilation resumes. The compiler has determined that `g` returns an `int32` value since the function `g` is annotated to indicate that it returns `int32` values.
- Execution resumes and `g` returns to the call continuation in `f`. The call continuation stub is executed.
- The call continuation in `f` is compiled. The compiler sees that `g` has been annotated as returning `int32` values. The code in `f` is optimized using this type information. No type check is performed at the addition.

The call continuation specialization process presented so far is able to optimize recursive calls in the `sumList` example and eliminate the type tag check on the return value. However, as explained in the next section, this process is speculative and does not work for every function.

4.3.1 Invalidating Call Continuations

The `makeList` function from Figure 4 is an example where the speculative call continuation specialization process fails. This is because `makeList` can return both objects and `null` values. As such, we cannot specialize callers of the function based on a single return type tag. In this situation, the speculative call continuation specialization process will try to specialize continuations, fail, and deoptimize them.

The first time that the `makeList` function returns, it will return a `null` value. This first return will then trigger the compilation of a specialized call continuation which assumes the return type of `makeList` to be `null`. When the function later returns a value with type tag `object`, this will be detected at code generation time. Callers of the `makeList` function will then have their call continuations deoptimized.

The deoptimization is done simply by writing stubs over already compiled call continuations. Should another `makeList` call return to a deoptimized call continuation, the stub will trigger the compilation of a new continuation. This time, the return type will not be specialized, because we know that `makeList` can return values with multiple type tags.

The speculative optimization and deoptimization process we employ could be seen as wasteful. We could have employed a static analysis instead. However, it can be difficult to establish the return type of a JS function simply by analyzing its code. Furthermore, the speculative approach can be more precise than a static analysis, because it is able to

take the run time behavior of code into account. The `return` statements which are never executed will not be taken into account. A static analysis does not know exactly which `return` statements are executed and which are not, but BBV does.

5 Evaluation

This section reports on an empirical evaluation of interprocedural basic block versioning. This evaluation was carried out based on an implementation of the extensions presented in this paper, namely typed shapes, entry point specialization, and call continuation specialization, within the Higgs JavaScript compiler.

A total of 26 industry benchmarks were selected from the SunSpider and V8 suites. The authors decided not to use the JSBench benchmarks [29] as they are more suited to fast interpreters (they are short running and have little computation). Benchmarks for which performance hinges on compiling regular expressions were omitted, as this is not a feature supported by the Higgs compiler.

To measure steady state execution time separately from compilation time in a manner compatible with Higgs, V8, SunSpider, TraceMonkey, and Truffle/JS, the benchmarks were modified so that they could be run in a loop. Warmup iterations are first performed so as to trigger JIT compilation and optimization of code before timing runs take place. Unless otherwise specified, 1000 warmup iterations and 100 timing iterations are used.

V8 version 3.29.66, SpiderMonkey version C40.0a1, TraceMonkey version 1.8.5+ and Truffle/JS v0.9 were used for performance comparisons. Tests were executed on a system equipped with an Intel Core i7-4771 CPU and 16GB of RAM running Ubuntu Linux 14.04. Dynamic CPU frequency scaling was disabled to ensure reliable timing measurements.

5.1 Method Identity

The extended version of Higgs tracks object shapes. Without them, the compiler would not be able to dermine which method is invoked at a call-site. With typed shapes, on average, the identity of the callee method is known for 90% of calls executed dynamically. When entry point versioning and call continuation specialization are performed, that number increases to 97.5% of calls. In practice, the identity of callees is known for most call sites. The exceptions are dominated by implementation limitations of the current version of Higgs, which currently treats captured closure variables as having unknown type.

5.2 Type Tests

Figure 11 shows the proportion of type tag tests eliminated with different variants of basic block versioning. These numbers measure actual tests executed at runtime rather than tests occuring in the program text. The first column (intra BBV) is the baseline, the number of tests that could be eliminated with plain intraprocedural basic block versioning [14]. The second column (typed shapes) shows the results obtained by adding support for typed object shapes. The third column (entry spec) adds entry point specialization, and lastly, the fourth column (entry+cont spec) adds call continuation specialization. On average, the baseline eliminates 61% of tests, typed shapes increases this to 79%. Entry point specialization improves the result to 89%. Finally, the addition of call continuation specialization allows the elimination of 94.3% of dynamic tests, and, in several cases, nearly 100%.

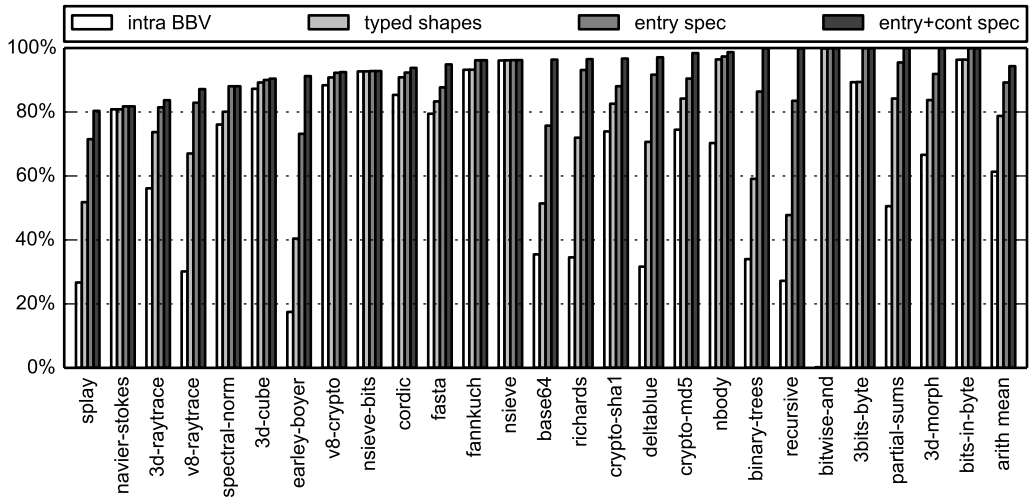

Figure 11 Proportion of type tests eliminated (higher is better).

5.3 Type Analysis

An obvious alternative to type propagation with interprocedural basic block versioning would be to perform a whole-program type analysis. As there are many different analyses in the literature with different degrees of precision, it is unclear how to evaluate the relative benefits of this paper's approach. It is possible to side-step the question by implementing an idealized static analysis. Each benchmark was run and the result of all tests was recorded. The benchmarks were then rerun with all type tests that always evaluate to the same result removed. The second run can be seen as an upper bound for the power of static analysis by itself. No static analysis can eliminate more tests than one that knows in advance the outcome of each of them. Figure 12 compares interprocedural basic block versioning and the idealized type analysis. The fact that basic block versioning outperforms type analysis should not come as a surprise. An analysis loses precision when control flow merges whereas basic block versioning creates separate versions to avoid this. The results suggest that no analysis can eliminate more than an average of 91.4% whereas Higgs can avoid executing 94.3% of tests. On more than half of the benchmarks, the proportion of eliminated tests exceeds 95%. In all benchmarks at least 80% of tests are removed.

5.4 Execution Time

The execution times of the benchmarks normalized to the unmodified version of the Higgs compiler [14] appear in Figure 13. With the exception of `navier-stokes`, `nsieve` and `nsieve-bits` (which are marginally slower), all benchmarks exhibit improvements. The largest speed up comes from the addition of typed object shapes, they improve execution time by an average of 26.8%. The addition of entry point specialization further improves performance, with a combined speedup of 36.3%. Finally, adding call continuation specialization brings the total improvement to 37.6%. The performance improvements brought by continuation specialization are relatively modest compared to those from entry point specialization. This is to be expected since entry point specialization allow us to eliminate more type tests (Section 5.2).

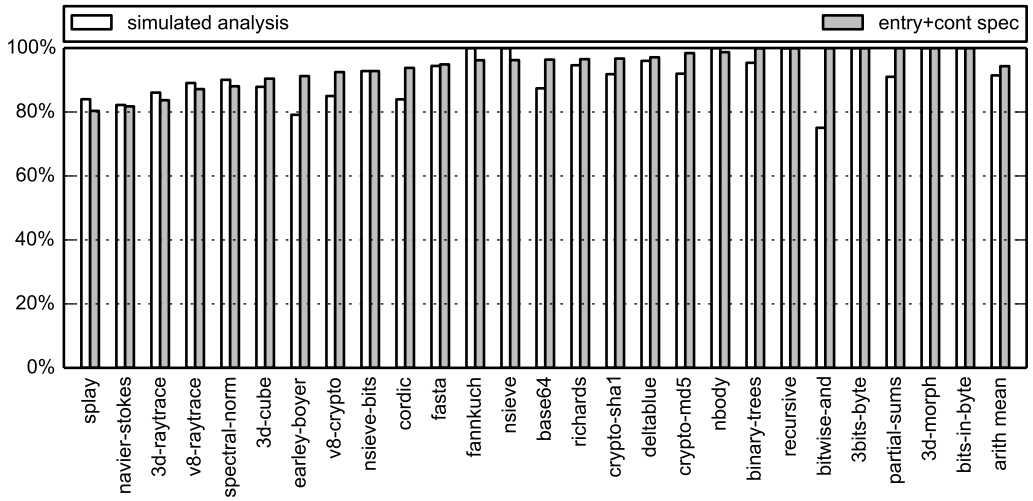

Figure 12 Proportion of type tests eliminated with BBV or a type analysis (higher is better).

5.5 Shape Tests

Our implementation of typed shapes is able to propagate known object shapes from one property access to another. There are many instances where multiple property reads on the same object occur within a given function, and shape propagation can allow eliminating further shape tests after the first property access on an object. Enabling typed shapes results in an average decrease of 27% in the number of shape tests over an unextended implementation of Higgs which uses untyped shapes and inline caches.

5.6 Call Continuation Specialization

Call continuation specialization uses a speculative strategy to propagate return type information. Call continuations for a given callee may be recompiled and deoptimized if values are returned which do not match previously encountered return types. Empirically, only 2% of functions executed cause the invalidation of call continuation code. Dynamically, the type tag of return values is successfully propagated and known to the caller 72% of the time. In over half of the benchmarks, the type tag of return values is known over 99% of the time.

5.7 Code Size and Compile Time

Adding entry point and continuation specialization to the unmodified Higgs compiler cause an increase in generated machine code size of 5.5% in the worst case and just 1.0% on average. Intuitively, one may have expected a bigger code size increase given that entry point versioning can generate multiple entry points per function. However, better optimized machine code tends to be more compact. Compile time increases by 3.7% in the worst case and just 0.01% on average.

5.8 Tracing Compilation

Tracing compilation bears important similarities to basic block versioning. One could expect tracing to do better because it can optimize long linear sequences of code. Tracing compilation

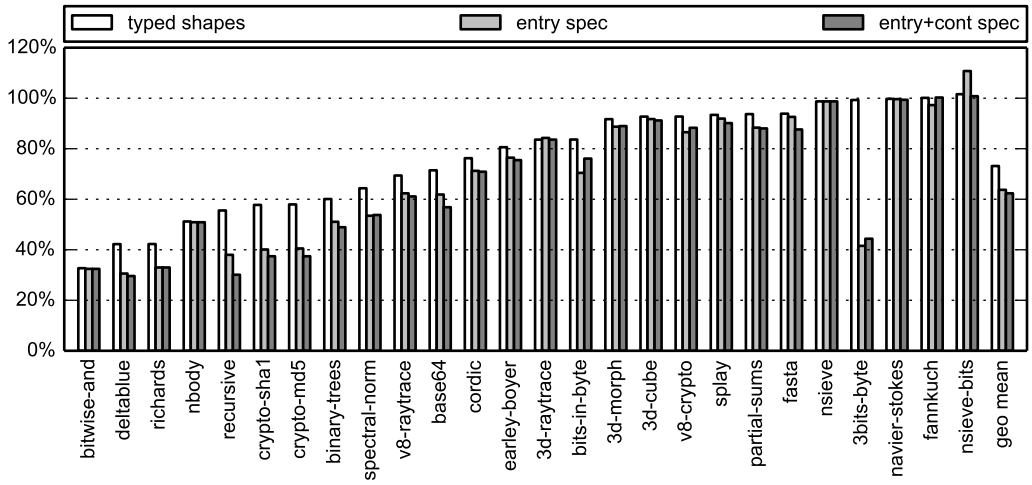

Figure 13 Execution time relative to baseline (lower is better).

was introduced to JavaScript with the TraceMonkey [17, 34] compiler. This compiler was in production within Mozilla's browser until 2011. Figure 14 compares the performance of the two compilers. On average, Higgs is 2.7x faster than TraceMonkey, and performs better on 22 out of 26 benchmarks. The benchmarks TraceMonkey achieves the best performance on tend to be ones which feature short and predictable loops.

The difference between the two is striking. It should be noted that TraceMonkey was built by a considerably larger team and implements strictly more optimizations than Higgs. For instance, it can inline while recording a trace. Even without inlining, Higgs does much better on the largest benchmarks. The two raytrace benchmarks, for instance, make significant use of object-oriented polymorphism and feature highly unpredictable conditional branches. The `earley-boyer` benchmark is the largest of all and features complex control-flow. The `splay` and `binary-trees` benchmarks apply recursive operations to tree data structures. We note that Higgs performs much better than TraceMonkey on the `recursive` microbenchmark which suggests TraceMonkey handles recursion poorly. While we caution against drawing definitive conclusions, it does appear that tracing compilation in the form implemented by TraceMonkey is mostly beneficial for computation with hot and predictable loops. Whereas Higgs is agnostic to the vagaries of control flow. It is worth mentioning that independent analysis of the behavior of real-world JavaScript programs suggests that hot and predictable loops are rare [30] and that TraceMonkey does not speed up real-world JavaScript programs such as the Google website [29].

5.9 Truffle/JS

Another interesting comparison is to look at the Truffle system from Oracle labs. Truffle/JS is an implementation of JavaScript written in Java and running on a modified Java virtual machine. Like Higgs, Truffle is a research prototype, but one being built by a larger team and with a code base about 6 times larger than Higgs'. It benefits from optimizations that are lacking in Higgs, such as method inlining and a sophisticated register allocator. For memory management it can defer to Java's highly tuned garbage collector.

Figure 15 shows the results of a performance comparison of Higgs against Truffle/JS.

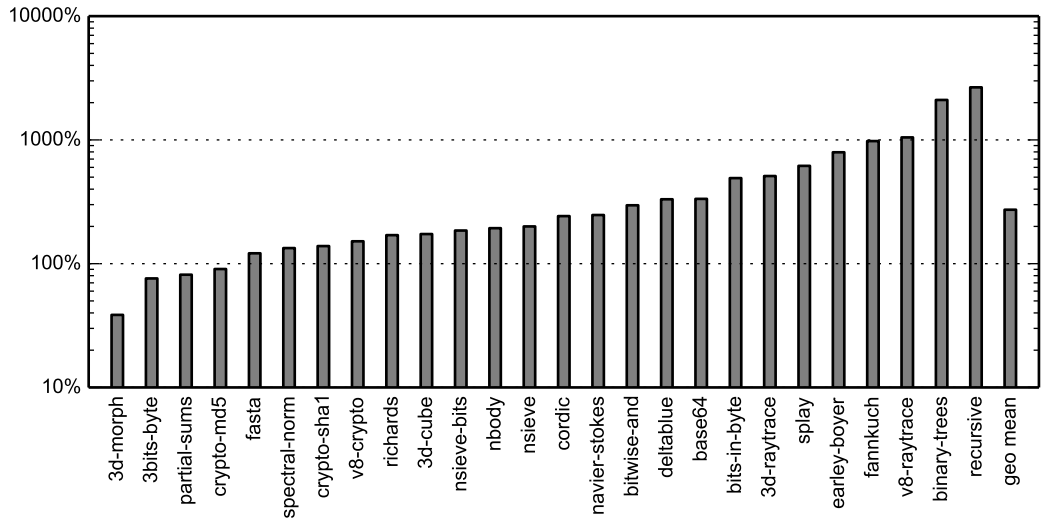

Figure 14 Speed relative to TraceMonkey (log scale, higher favors Higgs).

After both systems have gone through 1000 warmup iterations, Higgs is on average 69% as fast as Truffle/JS. The time recorded on the `3bits-byte` benchmark is zero, suggesting that Truffle used side effect analysis to optimize-away the computation.

Higgs and Truffle/JS, being research virtual machines, were not optimized for fast compilation. As a result, both systems are much slower than other engines when it comes to compilation times. We cannot directly measure the compilation time taken by Truffle/JS, but we can use the time it takes to warm up as a rough approximation.

Figure 16 shows the speed of Higgs relative to Truffle/JS when measuring the total time taken for 1000 iterations of our benchmarks, with no separate warmup iterations. On average, Higgs is 220% as fast as Truffle/JS on this comparison, indicating that the warmup and compilation time for Higgs is much shorter. This is not surprising, since Higgs begins generating type-specialized machine code as soon as program execution begins.

6 Conclusion

Basic block versioning is a compilation strategy for generating type-specialized machine code on the fly. This paper demonstrates how to extend this technique to propagate information across method call boundaries, both from callers to callees and from callees to callers, without requiring dynamic dispatch and without a separate type analysis pass.

Across 26 JavaScript benchmarks, interprocedural basic block versioning eliminates, on average, 94.3% of type tests. This is more than a static type analysis with access to perfect information could achieve. The proposed extension provides an average execution time reduction of 37.6% over an unextended basic block versioning implementation.

There is room for future work. While interprocedural basic block versioning yields encouraging results, more could be done. Two extensions to basic block versioning are planned: tracking types of closure variables and tracking array types. The Higgs compiler itself currently lacks several optimizations used by commercial virtual machines. While they are orthogonal to this paper, these optimizations may close the performance gap with commercial systems. The first optimization to add is method inlining. Inlining is likely

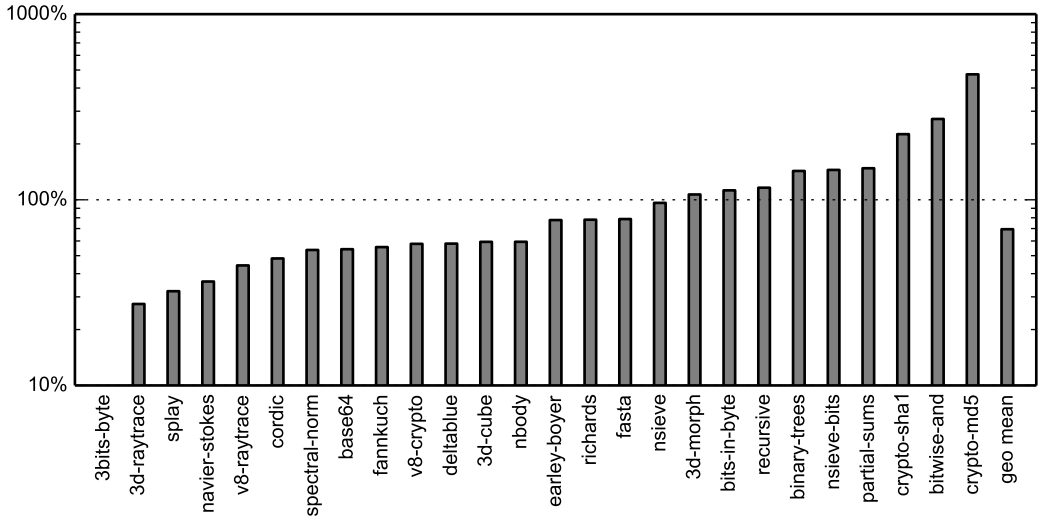

Figure 15 Speed relative to Truffle/JS (log scale, higher favors Higgs).

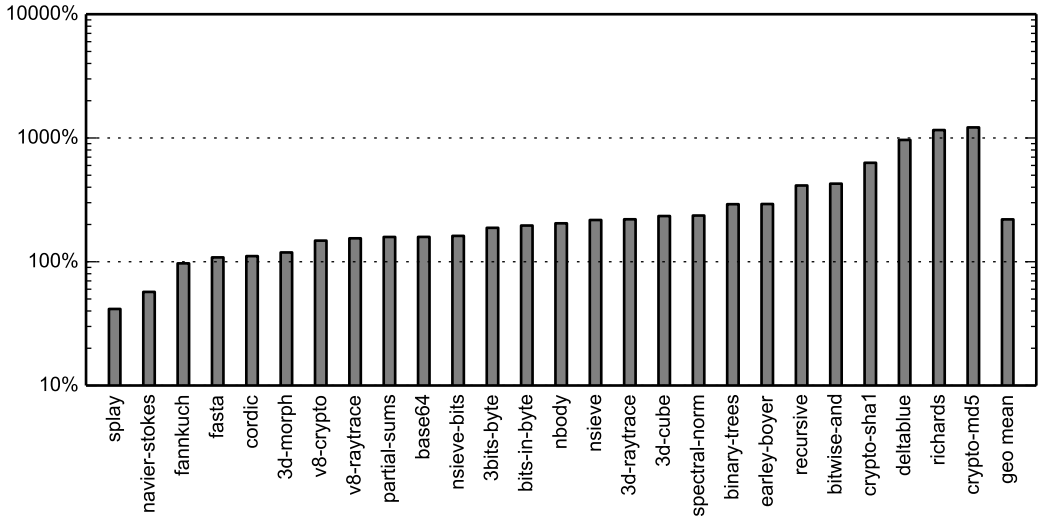

Figure 16 Speed relative to Truffle/JS, no warmup iterations (log scale, higher favors Higgs).

synergistic with basic block versioning as it provides more contextual information but it runs the risk of increasing code size as versions proliferate. Bloat can be mitigated by lazy, incremental, inlining where basic blocks are only added when needed. This would be faster than inlining entire control flow graphs without needing recompilation of the entire caller at inlining-time.

Acknowledgements. Special thanks go to Laurie Hendren, Jan Vitek, Erick Lavoie, Vincent Foley, Paul Khuong, Molly Everett, Brett Fraley and all those who have contributed to the development of Higgs.

References

1 Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. The HipHop virtual machine. In *Proceedings of the 2014 conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 777–790. ACM New York, 2014.

2 Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

3 George Almási and David Padua. MaJIC: compiling MATLAB for speed and responsiveness. In *Proceedings of the 2002 conference on Programming Language Design and Implementation (PLDI)*, pages 294–303. ACM New York, May 2002.

4 Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 Dynamic Languages Symposium (DLS)*, pages 53–64. ACM New York, 2007.

5 Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *Proceedings of ECOOP 2005*, pages 428–452. Springer Berlin Heidelberg, 2005.

6 V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the 2000 conference on Programming*, pages 1–12. ACM New York, 2000.

7 Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010. ACM.

8 Carl Friedrich Bolz, Antonio Cuni, Maciej FijaBkowski, Michael Leuschel, Samuele Pedroni, and Armin Rigo. Allocation removal by partial evaluation in a tracing JIT. In *Proceedings of the 20th ACM SIGPLAN workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 43–52. ACM New York, 2011.

9 Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: PyPy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

10 Carl Friedrich Bolz, Tobias Pape, Jeremy Siek, and Sam Tobin-Hochstadt. Meta-tracing makes a fast Racket. *Workshop on Dynamic Languages and Applications*, 2014.

11 C. Chambers, D. Ungar, and E. Lee. An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, September 1989.

12 Craig Chambers and David Ungar. Customization: optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *Proceedings of the 1989 conference on Programming Language Design and Implementation (PLDI)*, pages 146–160. ACM New York, June 1989.

13 Craig Chambers and David Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the 1990 conference on Programming Language Design and Implementation (PLDI)*, pages 150–164. ACM New York, 1990.

14 Maxime Chevalier-Boisvert and Marc Feeley. Simple and effective type check removal through lazy basic block versioning. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, volume 37 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 101–123. Schloss Dagstuhl, 2015. `http://arxiv.org/abs/1411.0352`.

15 Maxime Chevalier-Boisvert, Laurie Hendren, and Clark Verbrugge. Optimizing MATLAB through just-in-time specialization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 46–65. Springer Berlin Heidelberg, 2010.

16 Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1859–1866. ACM New York, 2009.

17 Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009.

18 Andreas Gal, Christian W. Probst, and Michael Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual Execution Environments (VEE)*, pages 144–153. ACM New York, 2006.

19 Michael Gorbovitski, Yanhong A. Liu, Scott D. Stoller, Tom Rothamel, and Tuncay K. Tekle. Alias analysis for optimization of dynamic languages. In *Proceedings of the 6th Symposium on Dynamic Languages*, DLS '10, pages 27–42, New York, NY, USA, 2010. ACM.

20 David Gudeman. Representing type information in dynamically typed languages, 1993.

21 Brian Hackett and Shu-yu Guo. Fast and precise hybrid type inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, pages 239–250. ACM New York, June 2012.

22 Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38, London, UK, UK, 1991. Springer-Verlag.

23 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*, pages 238–255. Springer Berlin Heidelberg, 2009.

24 Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Proceedings 17th International Static Analysis Symposium (SAS)*. Springer Berlin Heidelberg, September 2010.

25 Vineeth Kashyap, John Sarracino, John Wagner, Ben Wiedermann, and Ben Hardekopf. Type refinement for static analysis of JavaScript. In *Proceedings of the 2013 Dynamic Languages Symposium (DLS*. ACM New York, 2013.

26 Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. Improved type specialization for dynamic scripting languages. *SIGPLAN Not.*, 49(2):37–48, October 2013.

27 Francesco Logozzo and Herman Venter. RATA: rapid atomic type analysis by abstract interpretation; application to JavaScript optimization. In *Proceedings of the 2010 international conference on Compiler Construction (CC)*, pages 66–83. Springer Berlin Heidelberg, 2010.

28 John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing (LCPC)*, pages 566–580, 1995.

29 Gregor Richards, Andreas Gal, Brendan Eich, and Jan Vitek. Automated construction of JavaScript benchmarks. In *Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 677–694, 2011.

30 Gregor Richards, Sylvain Lesbrene, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, June 2010.

31 Armin Rigo. Representation-based just-in-time specialization and the Psyco prototype for Python. In *Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '04, pages 15–26, New York, NY, USA, 2004. ACM.

32 Baptiste Saleil and Marc Feeley. Code versioning and extremely lazy compilation of Scheme. In *Scheme and Functional Programming Workshop*, 2014.

33 Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. Just-in-time value specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–11, Washington, DC, USA, 2013. IEEE Computer Society.

34 Rodrigo Sol, Christophe Guillon, FernandoMagno Quintão Pereira, and Mariza A.S. Bigonha. Dynamic elimination of overflow tests in a trace compiler. In Jens Knoop, editor, *Proceedings of the 2011 international conference on Compiler Construction (CC)*, pages 2–21. Springer Berlin Heidelberg, 2011.

35 Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. An object storage model for the Truffle language implementation framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 133–144, New York, NY, USA, 2014. ACM.

36 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 2012 Dynamic Language Symposium (DLS)*, pages 73–82. ACM New York, 2012.

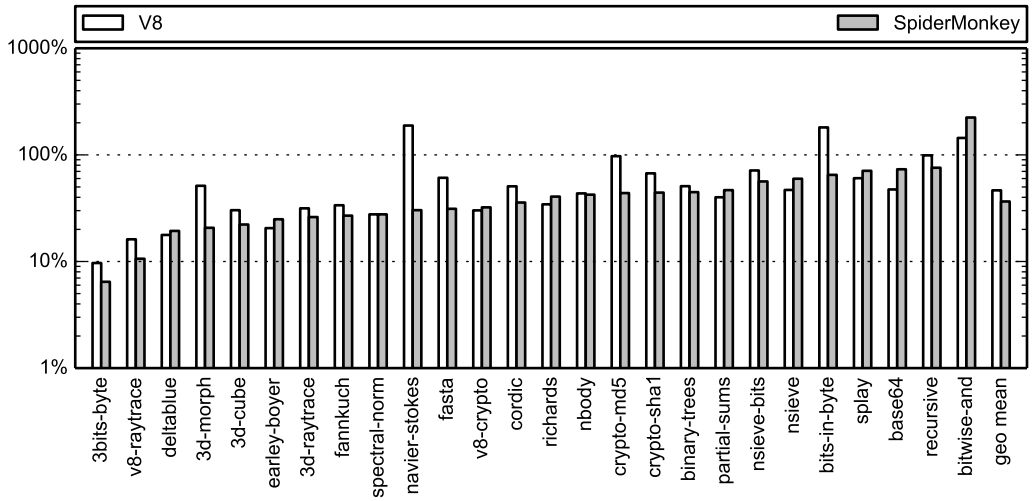

Figure 17 Speed of relative to commercial JS engines (log scale, higher favors Higgs).

A Comparison with V8 and SpiderMonkey

Figure 17 compares the speed of Higgs to optimized commercial JavaScript virtual machines. Higgs is generally slower, sometimes by an order of magnitude. There are a few benchmarks where it outperforms V8. Notably, `bits-in-byte` features many function calls, and Higgs is able to optimize this fairly well. The `bitwise-and` microbenchmark is also interesting because it is a loop performing global object property accesses. Higgs outperforms every JS engine we have tested on this benchmark, suggesting that it has faster global property accesses, thanks to typed shapes. On the other hand, Higgs is slower everywhere else. This is probably because Higgs lacks orthogonal optimizations such as: loop-invariant code motion, global value numbering, bounds check elimination, automatic SIMD vectorization, method inlining, allocation sinking, floating-point register allocation, etc. In the absence of these optimizations, BBV is most promising for use in a baseline JIT compiler.