Of JavaScript AOT Compilation Performance

MANUEL SERRANO, Inria/Université Côte d'Azur, France

The fastest JavaScript production implementations use just-in-time (JIT) compilation and the vast majority of academic publications about implementations of dynamic languages published during the last two decades focus on JIT compilation. This does not imply that static compilers (AoT) cannot be competitive; as comparatively little effort has been spent creating fast AoT JavaScript compilers, a scientific comparison is lacking. This paper presents the design and implementation of an AoT JavaScript compiler, focusing on a performance analysis. The paper reports on two experiments, one based on standard JavaScript benchmark suites and one based on 18 new benchmarks chosen for their diversity of styles, authors, sizes, provenance, and coverage of the language. The first experiment shows an advantage to JIT compilers, which is expected after the decades of effort that these compilers have paid to these very tests. The second shows more balanced results, as the AoT compiler generates programs that reach competitive speeds and that consume significantly less memory. The paper presents and evaluates techniques that we have either invented or adapted from other systems, to improve AoT JavaScript compilation.

CCS Concepts: • **Software and its engineering → Just-in-time compilers**; **Source code generation**; **Object oriented languages**; **Functional languages**.

Additional Key Words and Phrases: JavaScript, Scheme, AOT, JIT, Compiler, Dynamic Languages

ACM Reference Format:
Manuel Serrano. 2021. Of JavaScript AOT Compilation Performance. *Proc. ACM Program. Lang.* 5, ICFP, Article 70 (August 2021), 30 pages. https://doi.org/10.1145/3473575

1 INTRODUCTION

JavaScript is particularly difficult to implement efficiently because most of its expressions have all sorts of different meanings that involve all sorts of different executions that are not distinguished by any syntactic or type annotation. For instance, "`obj.prop`" might *i)* fetch property `prop` from `obj`, *ii)* scan the linked list of `obj`'s prototype chain and fetch `prop` from another object, *iii)* call a user defined function if `prop` is an accessor, *iv)* allocate a fresh object if `obj` is a primitive value, or *v)* evaluate yet another user function if `obj` is a proxy object. Checking all the possible interpretations and executing the appropriate one literally, that is treating the language specification as an algorithm, delivers unacceptably slow performance. All fast implementations use alternative strategies. Amongst all the possible interpretations, they favor the one that corresponds to the most frequent situation, for which they elaborate a faster execution plan, and, as importantly, for which they elaborate a fast guard that ensures the preservation of the language semantics. Typically, that is what *inline caches* and *hidden classes* achieve [Artoul 2015; Chambers and Ungar 1989; Chambers et al. 1989]. Using a single test, the comparison of the object's hidden class with the inline cache, we know if the property is to be read directly from the object and, if so, at which offset. The common intuition is that only dynamic compilers, *a.k.a.*, JIT compilers, can handle dynamic languages efficiently

Author's address: Manuel Serrano, Inria/Université Côte d'Azur, Inria Sophia Méditerranée, 2004 route des Lucioles, Sophia Antipolis, F-06902, France, Manuel.Serrano@inria.fr.

2475-1421/2021/8-ART70
https://doi.org/10.1145/3473575

because this heuristic-based strategy requires having the program *and* the data on hand in order to generate efficient code [Dot et al. 2017]. We view this position as too extreme, as it is oblivious to other AoT compiler characteristics that might make them competitive.

- AoT compilers can allocate conceptually infinite resources for analyzing and optimizing the program because they run before execution. This opens opportunities to conceive and deploy new optimizations that are out of reach of JIT compilers for which compilation time and compilation resource consumption matter.
- AoT compilers are efficient even for brief executions while JIT compilers need the execution to last sufficiently long to benefit from gathered profiled data. This should give AoT compilers an advantage for executing programs such as shell commands or cloud computing microservices [Wimmer et al. 2019].
- New techniques [Serrano 2018] have been proposed to adapt JIT-style heuristics-based approach to AoT compilation. These studies have shown that if indeed a JavaScript expression involves many different interpretations, typically only one is used over and over execution [Würthinger et al. 2017] and guessing it before the execution is not too difficult.

As few are committed to developping optimizing AoT JavaScript compilers, we rely too heavily on our intuition to answer the question whether AoT compilers can deliver performance comparable to JIT compilers. To provide the elements of a proper scientific comparison, we built Hopc, an AoT compiler for JavaScript. In this paper, we compare its performance with those of production JIT compilers and we show that on many new[1] tests, its performance is close to those of JIT compilers. We read this as a strong indication that an AoT compiler that optimizes the whole core language and the whole set of libraries could compete with the fastest JIT compilers. The paper presents this experiment, the overall architecture of the compiler, and some new optimizations and new implementations techniques.

The Hopc compiler **supports all of JavaScript 6** and many of recent JavaScript additions (proxies, `async`/`await`, generators, weak tables, modules, ...) with three restrictions: its regular expressions are not fully compliant with the ECMAScript specification, direct `eval` has no access to its lexical environment, and modules enforce a strict isolation of builtin global variables. Hopc passes **all tests of the 262 test suite** apart from those that rely on these very features.

Beside performance concerns there are other incentives for studying the AoT compilation of dynamic languages.

- JIT compilers require the target platform to support allocation of executable memory and they require the platform to offer enough memory for them to run side by side with the application, which practically rules out micro-controllers and significantly complicates supporting W^X platforms [Wikipedia 2021]. These limitations do not affect AoT compilers.
- New execution platforms may require AoT compilation. More precisely, if wasm [Haas et al. 2017] succeeds at becoming the universal execution machine, AoT compilation will likely impose itself because wasm currently lacks support for efficient dynamic code modification.
- From a different perspective, one might also wonder if consuming significant energy to compile the same code over and over again, as JIT compilers do, is a very sensible option, as energy consumption is the primary expense today. For instance, is it sensible that TypeScript

[1] See Section 3.1 for a rationale why new benchmarks make sense.

programmers who use the command line keep recompiling the same 100,000 JavaScript lines in the TypeScript compiler each time they modify their own application code?

The rest of the paper is organized as follows. Section 2 presents the related work. In Section 3, we present the experimental methodology we use for measuring and presenting the performance of the systems we compare. In Sections 4 and 5, we present the Hopc compiler architecture, its optimizations, some of its important data structures, and its back end. In Section 6, we present the performance evaluation report. We conclude in Section 7.

2 RELATED WORK

"The dynamic typing feature of [JavaScript, Python, ...] require that their applications are Just-in-Time (JIT) compiled" [Dot et al. 2017]. This sort of claim is so deeply rooted in the compilation and programming language communities that very little effort is invested in studying static compilation of these languages; JIT studies, however, are countless [Bauman et al. 2015; Brown et al. 2020; Chevalier-Boisvert and Feeley 2015, 2016; Choi et al. 2019; Gal et al. 2009a,b; Prokopec et al. 2019; Ren and Foster 2016; Saint-Amour and Guo 2015; Würthinger et al. 2017].

There are a few exceptions. Samsung's SJS [Chandra et al. 2016] was among the first attempts to build an AoT compiler for JavaScript. Its performance is very competitive but it restricts the language so drastically (the polymorphism was limited, the prototype chain was not fully supported, introspection and dynamic field lookup were lacking) that it is difficult to compare it with full JavaScript implementations. Static TypeScript [Ball et al. 2019], STS, pursues similar goals. It is a static compiler for a restricted TypeScript subset (no prototype support, no `arguments`, no `apply`, and builtin objects cannot be extended) designed to target 32bit micro-controllers. As with SJS, it shows very good performances but it also lacks too many dynamic features to be compared to hopc. Google V8 supports JIT-less compilation [Google 2019] but the lack of technical description makes it difficult to compare.

The literature about static analysis of JavaScript and Python is particularly abundant [Anderson et al. 2005; Arceri et al. 2020; Choi et al. 2015; Hackett and Guo 2012; Jensen et al. 2009a,b,c; Kashyap et al. 2014; Ko et al. 2015; Lerner et al. 2013; Logozzo and Venter 2010; Monat et al. 2020; Nielsen and Møller 2020; Park and Ryu 2015] but most of these studies are intended to build tools for static verification and they all assume analysis of global programs. They are not directly applicable to modular static compilers.

Profile-guided optimizations (PGO) are at the heart of JIT compilers. A. Wade *et al.* [Wade et al. 2017] measures the impact of PGO on JIT and AoT compilation of Java. This work is not directly applicable to JavaScript because the dynamicity of Java is nothing compared to JavaScript and also because the AoT compiler they test does not use optimistic compilation. The study [Choi et al. 2019] shows a PGO that improves the creation of hidden classes in JavaScript production executions, mostly for improving startup times. This could apply to AoT compilation as well. PGO has also been studied in the context of dedicated hardware support [Dot et al. 2017]. The application to our work is not straightforward.

Fortunately, other analyses and runtime techniques designed for JIT compilation apply equally well to AoT compilers. For instance, the study [Ahn et al. 2014] reports on the information that should be or should not be included in hidden classes. The dynamic allocation-size-based optimization [Clifford et al. 2015] applies equivalently well to AoT and it is used in hopc. The study [Qunaibit et al. 2018] presents MegaGuards, a system that automatically offloads part of Python programs into specialized hardware such as GPU. The stability analysis developed to satisfy the constraints imposed by these accelerators could potentially be applied to AoT compilation.

3 EVALUATION METHODOLOGY

In this study we are interested in comparing the performance of JIT and AoT compilers at a coarse grain level, in contrast with many studies that focus on specific optimizations, seeking specific acceleration. Accordingly, we propose a new benchmarking methodology and a new visual representation of the results. Instead of running a benchmark *n* times and presenting the arithmetic mean of the collected execution durations (as we do when only peak performance is important, see for instance Figure 5), we run the benchmark inside a loop and collect the execution times of successive runs as the number of iterations increases. See Figure 9 for an example. For the first run the almabench.js test has been executed once, twice for the second, etc.

This representation has advantages compared to a representation based on a single execution time per benchmark[2]. The curves of the plots provide a visual representation of the performance differences between systems, as barchart graphs would do. In addition, if a system suffers from a memory leak, its corresponding curve will be quadratic, something invisible if only one point of the curve is reported. If a system suffers a long warm up period, this will be reflected by a discontinuity in its curve. If a JIT optimizes and de-optimizes, this will be visible too. This representation will also show at which point JIT and AoT curves cross, if they do and when they do. Finally, this representation makes it easier to detect environment and configuration hazards [Mytkowicz et al. 2009], as that will generate visible discontinuity points.

3.1 JavaScript Benchmarking

The high-level abstraction of JavaScript makes the performance difference of efficiently optimized patterns and literally interpreted ones tremendous. Missing optimization opportunities can have dramatic consequences. We have observed situations where missing optimization opportunities incurred 10× or even 100× slowdowns. This makes benchmarking JavaScript compilers critical but also very difficult because it makes benchmarks *prescriptive*. They evaluate the performance but they also implicitly *define* or *characterize* the set of patterns compilers should favor. This duality creates a sort of conflict of interest: benchmarks simultaneously create the demand and its evaluation.

It is now well known and well documented that high scores have been obtained for *all* widely used JavaScript benchmark suites (JetStream, Sunspider, Kraken, Octane, ...) by observing and taking benefit of programs peculiarities that enabled compilers to deploy particularly efficient but also highly specific optimizations that seldom apply to general contexts or that can even hurt global performance [Meurer 2016; V8 Team 2017]. JavaScript benchmarks create a paradox. They are needed to improve performance, and history shows that they have accomplished this mission very successfully, but they also expose compilers to the risk of overfitting.

The JavaScript versatility is the second problem that makes it difficult to evaluate. How to find a set of representative JavaScript programs? Can such a thing even exist? Load time is a critical criterion for web client-side but unimportant for the server-side. On the other hand servers probably need fast long-running programs, something that will almost never be used on clients. JavaScript IoT device programming will also probably come with other requirements such as lean memory consumption and small code sizes. Some criteria are even be opposed to each other. For instance, it is probably impossible to simultaneously maximize cpu speed *and* minimize memory footprint.

The two problems posed by JavaScript benchmarks have been clearly identified by the community [Meurer 2016; V8 Team 2017] and the response has been to retire benchmark suites after a while and to propose new tests reflecting the evolution of the languages and its use. The two most recent suites are Speedometer and Webtooling. They address the overfitting problem by the mere

[2]Unconvinced readers will find a more traditional numerical presentation of the benchmarks results in the Appendix.

fact of being new and by increasing the language coverage. They address the second problem by segregating between web-client code and server code. None of these suites are well adapted to our experiment. Speedometer evaluates typical web-browser work load and Webtooling packages all the tests in a single very large file that does not fit with a modular compilation model. This is why we have assembled our own test suite for conducting our in-depth experiment. However, Octane, Sunspider, and JetStream are so extensively for evaluating JavaScript implementations, that although they come with the aforementioned biases, we also present the hopc score on these tests to give context and establish continuity with previous work.

3.2 Benchmark Suite

We have selected 18 different tests, written by different authors, using different programming styles. They cover a large part of the language's constructs and libraries but they are insufficient to show that one system is globally faster than another. As we only aim at showing that *a static compiler can deliver comparable performance on a significant portion* of the language, they fit our needs. For each program, we give its size, its allocation rate, the dominant allocation source, and a brief functional description.

(1) **almabench.js** [**410loc, 500MB/s (float 96%)**]: this is the translation of a C++ benchmark that calculates the daily ephemeris for the 21th century. It is a floating point intensive program.
(2) **bague.js** [**163loc, 1MB/s**]: this is the translation of a Scheme benchmark that solves the *Baguenaudier* game[3]. It uses simple recursive function calls, fixed integers, and array accesses.
(3) **basic.js** [**353loc, 280MB/s (string 80%, array 20%)**]: this is a tiny Basic interpreter whose original implementation [Eder 2020] has been adapted to match JavaScript strict-mode requirements and to replace the `eval` function originally used to evaluate Basic literals and variables.
(4) **boyer-scm.js** [**557loc, 210MB/s (object 97%)**]: Boyer.js and Earley.js are two tests that belong to the Octane test suite. They have been generated by the Hop Scheme->JavaScript compiler [Loitsch 2005; Loitsch and Serrano 2008]. To mitigate the overfitting bias described in [V8 Team 2017] we manually translated the original Scheme program into JavaScript by using the most neutral possible translation. Scheme functions are translated into JavaScript functions. Scheme pairs are encoded as instances of a JavaScript class with two fields. Arithmetic operators are mapped into JavaScript operators, etc.
(5) **earley-scm.js** [**1132loc, 160MB/s**]: this is the manually translated version of the earley.scm that Google used to produce Octane earley.js.
(6) **jpeg.js** [**2034loc, 600MB/s (array 100%)**]: this is a jpeg encoder/decoder [Incorporated 2020] used to encode and decode a 16×16 image. It uses bitwise operations and array read/write extensively.
(7) **js-of-ocaml.js** [**1652loc, 351MB/s (array 100%)**]: this is an OCaml game of life test compiled down to JavaScript by the js_of_caml compiler [Vouillon et al. 2020] and interpreted by the OCaml byte code interpreter compiled in JavaScript.
(8) **leval.js** [**450loc, 580MB/s (object 99%)**]: Leval.js is a Scheme interpreter. It first compiles expression into JavaScript closures and it then evaluates them by invoking the top-most closure. Activation frames are heap allocated using chained lists.
(9) **marked.js** [**2838loc, 160MS/s (strings 92%)**]: This is a Markdown parser [Jeffrey et al. 2020]. It reads its input from a string and generates a string containing the HTML translation.
(10) **maze.js** [**640loc, 236MB/s (object 47%, fun 33%)**]: This test implements a jigsaw game originally implemented in Scheme by O. Shivers. It is one of the few tests to use ES6 generators (for implementing a random generator).

[3] https://en.wikipedia.org/wiki/Baguenaudier

(11) **minimatch.js** [**1091loc, 300MB/s (string 53%, array 24%)**]: Minimatch.js is a popular npm package for file globing [Isaacs 2016] (22 million weekly downloads according to www.npmjs.com). It exercises string and array handling. This test is also one of the few tests that raises and catches exceptions.
(12) **minimist.js** [**1232loc, 238MB/s (array 58%, object 25%)**]: Minimist.js is the most popular npm package for command line parsing (37 million weekly downloads) [Halliday 2020]. The execution time is dominated by array constructions obtained by splitting strings.
(13) **moment.js** [**9321loc, 161MB/s (38% string, 30% array)**]: Moment.js is a time manipulation library [Moment.com 2020]. It is among the 10 most popular npm packages. This test is made from the core library and the first thousand tests.
Moment.js uses many different JavaScript features seldom used. For instance, it uses the `in` operator that checks the existence of a property in an object and the `delete` operator that removes a property. It uses expressions such as `var x = config._locale ||...`, a frequent pattern in real programs but surprisingly rare in standard benchmarks. Moment.js also uses library functions such as `hasOwnProperty` that also exercises hidden classes and inline caches in unusual ways.
(14) **qrcode.js** [**2295loc, 334MB/s (string 60%, real 27%)**]: This program [Kazuhiko 2009] builds QRcode. It first creates abstract representations of three different urls and then compiles them into svg, html, and ascii. This test allocates expensively short-living reals and strings and is a pattern that favors generational copying collectors.
(15) **richards+.js** [**532loc, 3MB/s**]: This is a modified version of Octane Richards.js where objects are wrapped with JavaScript proxies and all accesses go through proxy handlers. This is the only bechmark that tests proxy objects, introduced in JavaScript 6.
(16) **rho.js** [**3112loc, 391MB/s (array 31%, fun 31%)**]: This is a implementation of the Racket contract system [Strickland et al. 2012] in JavaScript. This implementation relies on the highly popular underscore.js library of high-level operations, which makes this test highly relevant.
(17) **uuid.js** [**1330loc, 300MB/s (string 40%, array 27%)**]: Uuid.js is another very popular npm package (35 million weekly downloads) [UUID 2020]. It computes universal unique identifiers. The original implementation uses Nodejs native support for `md5` and `sha1` which we have replaced with pure JavaScript implementations.
(18) **z80.js** [**976loc, 141MB/s (array 50%, fun 30%)**]: This is a Z80 emulator. It intensively tests integer operations.

We have developed another test suite made of synthetic micro benchmarks that we use internally to identify potential Hopc weaknesses and as an explanatory tool when studying benchmarks performance (see Section 6). All these micro-tests run a single loop that repeats the same basic operations (property accesses, function calls, Boolean tests, etc.). For the sake of space, this article only presents a subset of the whole set. We introduce them one by one when needed. To distinguish between the general performance tests and the micro-tests we denote the latter with a ⋆ mark.

All measures have been collected on Linux 5.3 x86_64, powered by Intel a Xeon E5-1650 processor. The tested production systems are : V8 (6.8.275.32), Jsc (4.0), and Js68 (C68).

4 THE HOPC COMPILER

JavaScript has grown into a *big* language. The 2018 edition of the specification [ECMA International 2018] counts 805 pages. The language supports a lot of core features: closures, prototype-based object orientation, class-based orientation, asynchronous programming, promises, generators, proxy objects, destructuring assignments, variable arity functions, etc. It also includes extensive libraries for arrays, numbers, regular expressions, strings, etc. To cope with this overwhelming

avalanche, the Hopc development strategy is two-fold. First, it does not directly generate assembly code but it generates Scheme code and it relies on Scheme optimizations for compiling efficiently the functional part of core JavaScript. This trades the complexity and the size of Hopc for less flexibility and less specialized data structures. We discuss the consequences of this strategy in the following sections. Second, Hopc supports all of JavaScript but it gives up optimizing some features and libraries. For instance, regular expressions use the `pcre` library whose performance is not always competitive [Herczeg 2015]. Fortunately, this will be easy to fix by replacing `pcre` with a more efficient implementation because this is an isolated component that has few connections with the rest of the system. More importantly, Hopc does not optimize floating point numbers, it boxes them and integer arithmetic operation overflows produce boxed floating point numbers. Contrary to regular expressions, optimizing floating point numbers is not an isolated optimization and we do not know if the techniques used by other systems (for instance, unboxed floating points stored in objects and arrays as V8 uses) would be easily doable and we do not know if they would impact the rest of the system. Our intuition is that we will be able to add floating point optimization without degrading the performance of integer operations but we have no evidence to back this up and this aspect is not evaluated in this paper. Finally, Hopc focuses on JavaScript *strict mode*, that is, it makes no attempt to optimize the deprecated `with` operator.

4.1 From JavaScript to Scheme, From Scheme to C

According to B. Eich, Scheme [Kelsey et al. 1998] is the source of inspiration for creating JavaScript [Flanagan 2002]. He conceived the language as a simplified version of Scheme, augmented with the Self prototype-based programming [Chambers and Ungar 1989], and using C syntax. The reminiscence of this early design is still observable: JavaScript is a true functional language with higher order functions and lexical scoping (although blurred by peculiar hoisting rules); it is dynamically typed; functions support variable number of arguments; variables and data structures are mutable. Many other features differ but the core JavaScript language [Guha et al. 2010] can be mapped naturally onto Scheme.

Hopc leverages decades of research on Scheme compilation to implement core JavaScript efficiently. Hopc compiles JavaScript modules (either ECMAScript 6 modules [ECMA International 2015] or Nodejs modules) into Scheme modules. These modules are compiled into C and then compiled into object files. The object files can be linked together to assemble a binary executable program or a shared library that can be loaded within an interactive read-eval-print loop.

Many compilers of high level languages target C instead of assembly code for portability and to take advantage of the continuous improvement to C optimizations. The full benefit of these optimizations is easier to obtain if the generated C code looks familiar to the C compiler (the C stack is used to implement function calls and the variables are mapped into C variables) but this forces the garbage collector to treat pointers conservatively. In the context of JavaScript compilation, it also prevents using techniques that modify the executable code dynamically, for instance, needed for possibly faster inline caches implementations [Chambers et al. 1989].

4.2 Targeting Scheme Code

Hopc is in charge of all the JavaScript specific optimizations and just that. It handles the creation of hidden classes and inline caches that enable fast object access. It implements a range analysis whose result is used to map arithmetic operations into Scheme fixnums or Scheme flonums operations. It deploys heuristics to duplicate and specialize code fragments for optimistic compilation. It implements several type analyses that help refine code specialization. Other optimizations are delegated to the Scheme compiler and to the Scheme runtime system. Hopc uses the Bigloo Scheme compiler [Serrano 1992] for its backend.

Targeting Scheme has greatly simplified the implementation. The functional core language, the automatic memory management, and the runtime support for polymorphic functions and values were all there since the first day. That said, Scheme comes with drawbacks.

- Scheme abstracts away many implementation details, which gives less freedom and less flexibility to Hopc for selecting optimal compilation schemas and data representations.
- Some Scheme idiosyncrasies demand strategies that are pointless for JavaScript. For instance, Scheme optimizes pairs but pairs are rarely used in JavaScript. In contrast, JavaScript objects are extensible (new properties can be added after an object has been created) and do not map naturally to any Scheme data structure.
- JavaScript functions are regular objects while Scheme functions are immutable values. In consequence, a JavaScript function is implemented by two distinct objects. This penalizes the memory management and general function invocations.
- Scheme and JavaScript variable arity functions differ radically. In Scheme they are statically declared and they receive optional arguments in lists. In JavaScript all functions can be invoked with any number of arguments and the arguments are packed into an array-like structure called `arguments`.

To mitigate these problems we have added new optimizations and functionality to the Scheme underlying implementation. These additions are presented in Section 5.

This approach comes with an important limitation. The JavaScript-to-Scheme compilation typically expands the source file by between 5× and 10× and the Scheme-to-C compilation expands the intermediate Scheme file by about 10×, which means the whole JavaScript-to-C produces huge C files. Fortunately, the expansion mostly comes from the introduction of many temporaries that are eventually removed by the C register allocation. Hence the generated binary files remain small (see Section 6.4, Figure 10) but in practice, this limits the size of the files Hopc can compile because above a certain threshold, compilation times become excessively long. For instance, compiling minimit.js takes about 10 seconds on our experiment platform, but compiling moment.js lasts about 5 minutes! In practice, 10,000 to 15,000 lines of code is the maximum file size one can reasonably compile with Hopc. Note however, that this is not the maximal number of the lines of code of a whole application as Hopc supports modules and separate compilation.

4.3 JavaScript Front-End Compiler

Hopc compiles a JavaScript module into a Scheme module. This involves more than 50 different passes. Excluding the mandatory source code parsing, the symbols resolution, some mandatory JavaScript syntactic analyses (for instance *strict mode* detection) and the Scheme code generation, there remain about 45 passes of analyses and optimizations. As this paper focuses on the Hopc runtime system more than on the compiler internals, the compilation passes are described here only very succinctly. Readers interested in those details may refer to previous publications. [Serrano 2018; Serrano and Feeley 2019; Serrano and Findler 2020].

The first group (11 passes) contains the optimizations that are used to map the JavaScript statement-based syntax into the expression-based Scheme syntax and to efficiently handle JavaScript lexical scope. Although these optimizations do not involve innovative research results, they are crucial for generating quality code. In particular, naive handling of JavaScript lexical scope has a dramatic impact on the generated code. It is of the highest importance that the compiler can avoid the double initialization (a lifted `undefined` initialization followed by an assignment) of `var` declarations and the *dead-zone* dynamic check of `let` and `const` declarations, as these ruin the compiler type analyses and clutter the generated code with many, mostly useless tests.

The second group (10 passes) is about type analyses. The dynamicity of the language makes a single view of a variable's types and values insufficient. Generally the compiler is only able to establish that a variable is *likely* to be of a certain type or that it is of a particular type only for some statements and expressions. Using different ad hoc heuristics, the compiler can then decide to either duplicate program fragments for generating fast paths and slow paths or it can force the generated code to produce values of a certain type. The latter case mostly applies to numerical expressions. The JavaScript specification includes only floating point numbers but all fast implementations map these numbers to small fixnum integers as much as possible. The main challenge is then to mix efficiently optimized fixnum representations and full-fledged double representations in presence of possible arithmetic overflows. For instance, consider the arithmetic expression "`(`e_1`-1)+Math.cos(`e_2`)`". Even if the compiler can prove that e_1 is an integer, the numerical constant `1` and the expression "e_1`-1`" are better handled as floating point numbers in order to prepare the floating point addition with `Math.cos()`. These types are assigned using a bottom-up type propagation of numerical values. When one branch of a binary operation is a floating point, this type is propagated to the other branch and this process repeats until the fixed point is reached. This group contains several such optimizations. Taken separately, they do not radically change the performance of the compiler but combined all together they are absolutely central to deciding which fragments of the program to specialize and to optimize aggressively.

The third group (23 passes) contains the other optimizations such as inlining, dead-code removal, escape analysis, pre-allocation of inline caches, life-time analysis for optimizing the `arguments` pseudo-variable, loop analysis for faster array accesses, constructor size estimation, common sub-expression elimination that requires fine knowledge of JavaScript effects, constant lifting that enables sharing read-only objects, etc.

4.4 Object Representation

JavaScript uses two orthogonal type hierarchies: the primitive types such as `Object`, `String`, `Function`, `Array`, etc., and the classes that the program defines using `class` constructors or prototype chains. A JavaScript object can change its JavaScript class but it cannot change its primitive type. That is, a function cannot become an array, a string cannot become a regular expression, etc. A value can, however change its prototype chain and an object can be extended with new properties or shrunk by removing proprties. As these operations have no direct Scheme counterpart we detail here how objects are represented and manipulated.

The Bigloo compiler extends Scheme with single inheritance classes that are used to implement JavaScript primitive types. The `JsObject` class is the root of the JavaScript class hierarchy. Figure 1 presents the memory layout of the JavaScript object "`{x:5.66, y:-4.1}`". The `__proto__` field implements the JavaScript prototype chain. The `cmap` field is used for implementing the hidden classes and inline caches [Chambers and Ungar 1989]. It points to a map that associates property names with offsets. In the figure 1 we see that the property `x` is associated with offset 0, as it is the first element of the list, and property `y` is at offset 1, because it is the second element. The `elements` field points to a Scheme vector that stores the values of the object attributes.

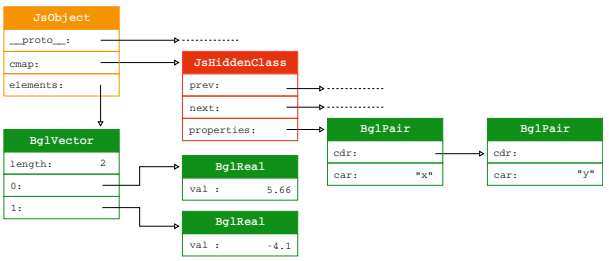


Fig. 1. The Hopc memory representation of the JavaScript Object "`{x:5.66, y:-4.1}`".

Floating point values are boxed allocated numbers. Vectors are represented as a contiguous memory chunk following a memory word for the vector length. The encoding of the type tag of these values is architecture dependent. Considering 64-bit platforms, pointer alignment gives the opportunity to use the 3 least significant bits of a memory word. Bigloo uses that opportunity to encode the type of pairs, reals, and vectors directly in the pointers, avoiding an extra header word. Objects (class instances) always contain a header field. It encodes a reference to their class (a 12-bit wide integer indexing entries of the global class tables) and leaves 16 bits available for user needs. These bits are used in Hopc to encode object JavaScript properties such as frozen objects, sealed objects and to implement some optimizations. For instance, one bit tells if an object owns a getter property or if all its properties are plain values. This enables faster property assignments.

A static analysis pre-computes the vector length of each constructor so when objects are allocated, the `elements` vector is large enough to hold all the properties that are added in the constructor body. If the object is later extended, the constructor vector length is updated [Clifford et al. 2015] so that the next allocated objects are large enough to avoid the dynamic extension. Hopc extends this technique to literal objects that do not have true dedicated constructors.

As shown in Figure 1, JavaScript objects are fat and accessing properties is slow. In comparison to Scheme objects, they need 2 extra header words and one indirection through the `elements` to access properties. To avoid the property indirection, Hopc uses a customized native allocation procedure to create *inlined* objects. Most fast JavaScript implementations use similar techniques. Instead of allocating separately the Bigloo `JsObject` and the Bigloo vector `BglVector`, a single chunk of memory is allocated and the `elements` attributes point to the object itself, as shown in Figure 2. Properties of inlined objects are accessed directly from the object pointers without reading the `elements` pointers. When an inlined object is extended, a new vector is allocated and the `elements` pointers updated. Inlined objects and plain objects are associated with different hidden classes so the inline cache test can check with a simple comparison of the type of an object and its inlinedness.

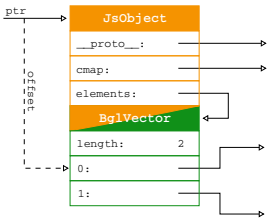


Fig. 2. Inlined object representation. Inline object properties are fetched with a single memory read from the object base pointer with an expression such as `*(ptr+offset)`, which improves over non-inlined objects that require two memory reads such as in `ptr->elements[x]`.

Figure 3 shows the effectiveness of this technique. The *inl. access* row reports the number of accesses that hit inlined properties. The *reg. access* row shows the percentage of regular non-inlined property accesses. The *cache miss* row shows the overall number of cache misses. The *realloc* row shows the percentage of objects that need to be extended after having been created. The *inl. access* and *reg. access* only consider property accesses that hit an inline cache and that access properties directly located in the object itself. They do not account for properties fetched from the prototype chain nor, of course, the properties not found at all. This is why summing inlined and non-inlined accesses is generally less than 100%.

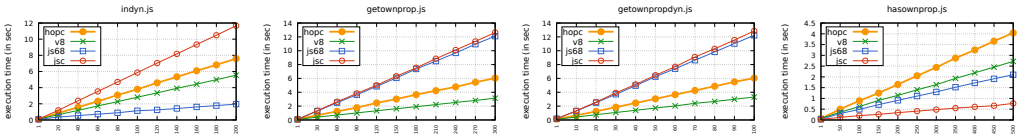
This experiment shows that tests do not need reallocation and that they seldom use non-inlined property accesses and that the encoding proposed in Figure 2 accelerates the vast majority of executions.

	almabench	bague	basic	boyer-scm	earley-scm	jpeg	js-of-ocaml	leval	marked	maze	minimatch	minimist	moment	qrcode	rho	richards+	uuid	z80
inl. access	0%	0%	0%	100%	100%	61%	1%	100%	66%	98%	63%	26%	33%	0%	13%	50%	31%	39%
reg. access	0%	0%	0%	0%	0%	0%	0%	0%	8%	1%	3%	0%	7%	0%	1%	0%	6%	0%
cache miss	34%	88%	52%	0%	0%	0%	0%	0%	11%	0%	0%	24%	8%	2%	22%	4%	12%	42%
extension	3%	3%	3%	0%	0%	2%	1%	0%	3%	0%	1%	0%	2%	3%	3%	2%	2%	0%

Fig. 3. Statistics about object property accesses and object re-allocations.

4.5 Property Accesses

Accessing object properties in prototype based languages is a difficult but crucial part of an implementation. Hopc techniques extend previous work on inline caches and hidden classes [Chambers and Ungar 1989; Hölzle et al. 1991]. They have already been presented in previous publications [Serrano and Feeley 2019; Serrano and Findler 2020] and are not repeated here. Here, we simply note that for JavaScript, it is also important that these techniques apply well both to direct property accesses and to the operators and library functions that manipulate properties (the `in` operator, `hasOwnProperty` that checks if an object owns a property, etc.). The following four micro-benchmarks evaluate these features separately. They show that Hopc is competitive for these operations.



Another important aspect to handle efficiently is access failures that happen when accessing missing properties. It involves caching failures and is subtle because the extension of a seemingly unrelated object might change many prototype chains and might then invalidate such negative caching. For that reason, caching any property that is not directly in an object, either because the property is absent from the object or because it is owned by an object of the prototype chain, requires some form of invalidation. For instance, consider the following fragment:

```
function check(v, p) { return p in v }
let p = {}, o = { __proto__: p, a: 1 };
check(o, "b"); // returns false;
p.b = 3;
check(o, "b"); // returns true, although `o` has not changed.
```

the assignment `p.b = 3` must invalidate the cache used for the `in` operator of the `check` function.

For this invalidation, Hopc uses a simple but effective technique that, to the best of our knowledge, is not described elsewhere. When the hidden class of any object belonging to a prototype chain is changed, all the caches that do not refer to properties directly owned by objects are invalidated. This might seem to be a drastic solution but it is efficient because prototype objects are generally only modified during the program initialization and remain constant for the rest of the execution. The question of detecting that an object belongs to a prototype chain remains, however. This is done at no cost by merely marking objects that are traversed once in the routines implementing cache misses. Hopc uses a conservative approach: as soon as an object is marked, it is assumed to belong to a prototype chain forever and any further extension or deletion of that object will invalidate the inline caches that denote not-directly-owned properties. This technique makes it useless to store prototype objects inside hidden classes that would increase the level of polymorphism of the programs, and that would then yield to additional cache failures [Ahn et al. 2014].

4.6 Functions and Function Calls

A JavaScript function is compiled into two objects: a JavaScript object and a Scheme procedure. Figure 4 shows the representation of `CTOR` closures created by the function `F`:

```
function F() {
   let cnt=0; let base=-4.1;
   return function CTOR(v) {this.val=v+base; this.cnt=cnt++}
}
```

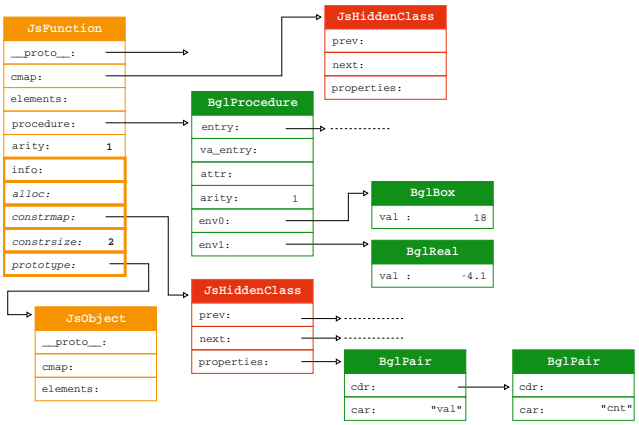


Fig. 4. JavaScript function encoding

In addition to the regular attributes shared by all JavaScript objects, the function object stores the JavaScript function `arity` that is used to adjust the formal parameters and the actual values, when values are missing or unexpected. The `info` attribute is a static structure that contains the function name and the text of its source code. The `alloc` attribute is a Scheme function that creates fresh objects when this function is used as a constructor. This is needed to distinguish between the constructors that create different builtin objects. For instance, the `RegExp` constructor allocates a `JsRegExp` object instead of a plain `JsObject`, the `Array` constructor function allocates `JsArray` objects, the `Function` allocates a `JsFunction` object, etc. The `constrmap` field points to the initial hidden class of the freshly created object (that is different from the function hidden class itself). The `constrsize` field holds the size of the object to be allocated on a `new` call (see Section 4.4). Finally, the `prototype` field is the JavaScript function prototype.

A Bigloo procedure has two entry points, which are C pointers to functions. In the case of JavaScript only the first one, `entry`, is used, and the second entry point is wasted. The `attr` field is used to distinguish the various type of functions Scheme supports. This is also unused by JavaScript. The procedure contains its own arity field, which differs from the JavaScript arity. The Scheme arity tells how many values must be passed in contrast with the JavaScript arity that tells how to pass values, and how to complement or how to remove extra parameters. Finally, the `envXXX` attributes store the closed free variables. Mutated closed variables are boxed (`cnt` in the example).

Encoding functions with one JavaScript object and one Scheme procedure is not memory efficient. To mitigate that problem, Hopc deploys various analyses to optimize function representations. Functions that are never used as values (never passed around to other functions nor stored in variables and data structures) are not allocated at all. Two strategies are used for those used as values. First, closures use stack allocation, as described in Section 5.5 and second, Hopc applies a

0cfa analysis [Shivers 1988] to the whole module. When a function does not escape, *i.e.,* when it is never stored into an object nor passed to an external function, and when all the functions that show up at use sites where this function is called satisfy the same properties, then the compiler generates only a Scheme closure, avoiding the entire JavaScript object. When the analysis shows that a function can never be used as a constructor, only a lightweight function is created (the italic attributes in Figure 4 are not allocated). These sorts of optimizations are relatively standard in compilers for functional languages but as they involve global fix-points over the whole compilation unit, they are difficult to use in JIT compilers, simply becuase a JIT compiler must limit compilation time and memory use. We show in Section 6 that these techniques let Hopc optimize closures effectively.

5 THE SCHEME BACKEND COMPILER

The Bigloo Scheme compiler maps Scheme functions to C functions and Scheme variables to C variables. It uses the Boehm-Weiser collector [Boehm and Weiser 1988] hence it can exchange Scheme data and C data without restriction and without any conversion. It supports optional type annotations that can denote Scheme values as well as C values. Finally, it supports low-level pragma expressions that enable expert programmers to tweak the generated C code. Hopc uses all these features to fine-tune the code it generates but this has shown to be insufficient to get competitive performances. We have had to add new optimizations, new runtime support, and a new allocation mechanism. They are presented in this section.

5.1 Numbers

Numbers are among the main pitfalls a JavaScript implementation must avoid. The ECMAScript specification defines numbers as "double precision floating point numbers", *i.e.*, 64-bit format IEEE-754 numbers. In other words, JavaScript has no fix precision integers (henceforth fixnums)! However, integers play central role in most programs because they are used to index arrays and because they are used in bit-wise operations. All fast implementations distinguish floating point numbers and integers but they rely on different strategies.

NaN Tagging. All values, including objects, are represented as floating point numbers. The most significant bits are used to distinguish value types. Two ingredients are needed for this encoding: the encoding of Not-a-Number values (NaN) and limited address space. The IEEE-754 specification accepts $2^{52} - 1$ different NaN values but actually the hardware and the software only use two: signaling and quiet NaN. The other values can be freely used by the application, in particular to represent allocated values because on current common 64 bit platforms the address space is restricted to the range $[0..2^{48}]$. This leaves 4 bits of the NaN values to encode value types. SpiderMonkey [MDN 2019; Mozilla 2020] and JavaScriptCore [Apple 2018] use this encoding.

Box/Smi. Floating point numbers are allocated (boxed) values and *SM*all *I*ntegers in the range $[-2^{32}..2^{32}-1]$ are presented as exact 32-bit values encoded in 64-bit wide word. The least significant bits of pointers are used to distinguish value types. The actual integer values can be stored in the lower or upper bits. The representation combines fast integer operations and fast integer overflow detection that is implemented by the hardware, especially if the integer value is stored in the upper 32 bits. This schema enables fast array indexes but it is potentially not suitable for string indexes that span over the range $[0..2^{53}]$. This number representation is used by Google's V8 [Google 2018].

Box/Int53. Floating point numbers are allocated (boxed) values and integers in the range $[-2^{53}..2^{53}]$ are stored as exact 64 bit values. The least significant bits of words are used to distinguish value types. Integer overflow detection is as fast as Box/Smi. Bit shift operations need masking but

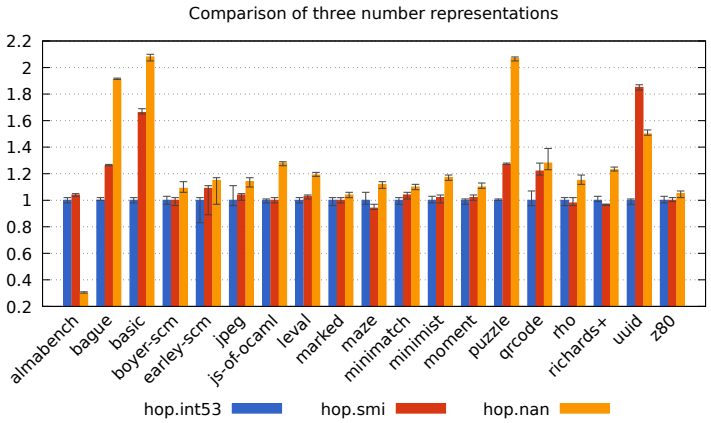


Fig. 5. Impact of the numbers implementation. This experiment compares the performance of Hopc where only integer and real representations differ. The vertical axis is the clock wall execution time relative to int53, using a linear scale. The measures have been collected on Linux 5.3 x86_64, powered by an Intel Xeon E5-1650.

the impact on overall performances is unnoticeable. On the other hand this schema enables fast indexing of arrays and strings and a peculiar property of IEEE-754 enables the compiler to avoid polymorphic detection when incrementing and decrementing indexes. Above 2^{53} only even integers can be represented. Hence, according to the IEEE-754 arithmetic $2^{53}+1 = 2^{53}$! A similar observation applies to negative values. In consequence, when the compiler generates a loop with an index ranging from 0 and incremented by 1, no matter the upper limit of the loop, the index will stick to the range $[0..2^{53}]$ and thus can always be represented as a 64 bit integer. This does not remove the need for the overflow check but this enables the compiler to use unboxed representation for indexes, which speeds up array access. To the best of our knowledge, Hopc is the only compiler to take advantage of this observation.

Bigloo supported only tagged int64 integers and boxed floating point numbers. We have extended it to support also NaN Tagging, Box/Smi, and Box/Int53 and we have compared the performance of these three representations using our benchmarks suite (Section 3.2). The results are presented Figure 5.

NaN Tagging delivers best performance for floating point intensive applications (almabench.js) but it is consistently slower for the other tests. This is because the tags encoding value types are stored in the most significant bits of values and manipulating these most significant bits of 64-bit words is slower than manipulating the least significant bits because integer literals exceeding 2^{32} cannot be embedded inside assembly instructions. As we deemed that floating point intensive application are less common in JavaScript, we have decided not to use NaN tagging for Hopc. In addition, we also observe that V8 do not use NaN but still seems capable of good performance on floating point applications (see Figure 9). So probably Hopc could enjoy much better performances too for these programs without changing its number encoding. Box/Smi and Box/Int53 deliver comparable performances on many tests but on some benchmarks that use many loops traversing arrays and that benefit from the non-overflowing IEEE-754 integer increment (bague.js, puzzle.js, and sieve.js), Box/Int53 shows performances significantly better. In addition, Box/Int53 collaborates more fruitfully with the Hopc range analysis. For instance, when the compiler proves that an integer in this the range $[-1..2^{32}-2]$, as it happens when a loop walks an array with decreasing indexes, the Box/Int32 representation is not large enough to ensure that there cannot be any arithmetic

overflow while Box/Int53 is. This gives Box/In53 a significant advantage and this is why we have opted for that encoding.

5.2 Tweaking the Generated C Code

Inline caches and hidden classes are central elements for good JavaScript performance. They are implemented as a set of Scheme macros. Assuming for simplicity that it can prove that `o` is an object, Hopc compiles a property access "`o.x`" into:

```
(js-jsobject-get-name/cache o (& "x") %this (js-pcache-ref %pcache 8))
```

The macro `js-jsobject-get-name/cache` expands into a hidden class comparison, a fast property access on success, and a call to the cache miss routine on failure. The operator "`&`" is a Scheme macro that transforms its argument (here the string "`x`") into a unique property name. The argument `%this` designates the JavaScript global object. The expression `(js-pcache-ref %pcache 8)` is another macro call that expands into a reference to an inline cache (here the 8^{th} inline cache of the module). Assuming that Bigloo compiles an access to a Scheme object property as a C structure access, the crucial part of "`o.x`" is the comparison of `o`'s `cmap` (see Section 4.4) and the cache entry. If `%pcache` was a regular Scheme variable, the comparison would be compiled into:

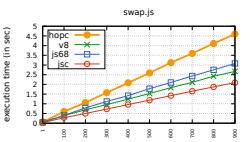
```
obj_t pcache; ... o->cmap === pcache[8]->cmap ...
```

as `pcache` would be a pointer to an allocated memory, `pcache[8]->cmap` would involve two memory accesses. To save one, `%pcache` is rather compiled as a C statically allocated array:

```
struct { void *cmap, ... } pcache[234]; ... o->cmap === &(pcache[8])->cmap ...
```

The C compiler compiles `pcache[8]` with a single memory access because it knows at compile time (or at link time) the static address of `pcache`.

The Bigloo compiler cannot do this transformation automatically but by the means of its `pragma` form, Hopc implements it. Here is the implementation of the `js-pcache-ref` macro:




```
(define-macro (js-pcache-ref pcache idx)
   `(pragma "&(pcache[$1])" ,idx))
```

Using this macro, the implementation of property accesses is as fast as C can be but it is slower than the assembly in-instruction address modification JIT compilers apply [Chambers et al. 1989]. Instead of storing and loading the property index inside the cache, they *inline* it as a shift operand of the load assembly instruction. The penalty of not using *full* inline caches is visible on the swap.js* benchmark that repeatedly reads and writes properties from two objects. As reading and assigning properties is a central operation, this penalty obviously degrades the overall Hopc performance. However, one should note that this is not a consequence of the AoT compilation principle but rather a consequence of targeting C instead of an assembly language. If Hopc was generating assembly code it could benefit from the inline cache technique too.

5.3 Local Exits

JavaScript is statement-based while Scheme is expression-based. JavaScript then supports the `return`, `break`, and `continue` keywords that have no direct counterparts in Scheme. Obviously they could be implemented using Scheme's `call/cc` but `call/cc` is difficult to implement efficiently, specially when compiling to C.

Bigloo provides an alternative to `call/cc` by the means of `bind-exit` that creates restricted continuations that can be used only in their dynamic extents. Initially `bind-exit` was compiled

using C `setjmp/longjmp` but this is not fast enough to fit Hopc needs that generates `bind-exit` for all functions that use a non-terminal `return` and for all loops that either use `break` or `continue`.

Contrasting with `call/cc` which is a regular function, `bind-exit` is a special form. This lets the compiler track when and how it is used and this makes it possible to optimize it. We have used this possibility to add a new optimization that works in two steps. First it selects all the continuations that are never used as first class values. Second, once the closure analysis has split and lifted functions, it replaces with direct `goto` the continuations that are syntactically located in the function that creates them. Example:

```
1  (define (F x)
2    (bind-exit (return)
3       (define (g y)
4          (return (+ x y)))
5       (g x)))
```

```
6  (define (H x)
7    (bind-exit (return)
8       (define (g y)
9          (return (+ x y)))
10         x)
11      (I g))
```

Inside `F` the `return` continuation is compiled as simple C `goto` (or `return`) because `g` is inlined and is not compiled as a separated C function. In `H`, `return` cannot be optimized because as `g` escapes, it is compiled as a separated C function and then the call to `return` line 9, once compiled, is executed outside the function `H` that created the continuation line 7. Fortunately, in the code Hopc generates, `return` always designated the closest function and `break`, and `continue` the closed loops so they never escape and they are always optimized.

5.4 Heap Allocation

Bigloo uses the Boehm & Weiser's collector [Boehm and Weiser 1988]. It is a mark & sweep collector using ambiguous roots. These characteristics prevent it from moving objects. Collection time is then proportional to allocated objects and not live objects contrary to moving collectors. This impacts negatively the performance of all tests that allocate many short living objects. It also impacts negatively allocation speed as it prevents bump allocation because collected dead objects are stored in free lists. This is particularly critical for Hopc that allocates two Scheme objects per JavaScript object.

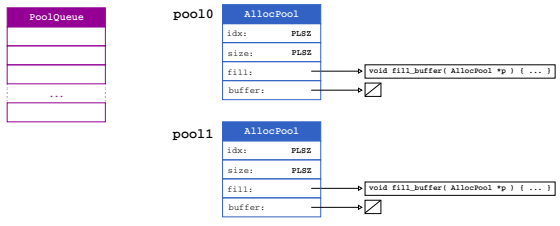
As presented in Section 4.4 (Figures 1 and 2) JavaScript objects have a complex structure. All objects, even those without properties, have at least 4 memory words to initialize: `__proto__` for the prototype chain, `cmap` of the object hidden class, `elements` that points to the possibly empty properties array, and `length`, the property array size. This initialization that takes place at each object creation requires a significant execution time.

To mitigate the slowness of object allocation and to speed up object initialization, Hopc implements a parallel allocator backed by the standard unmodified Boehm & Weiser allocation. The acceleration comes from the parallelization of the object pre-allocation and object initialization. This allocation schema only improves performance on multi-core architecture. It works as follows.

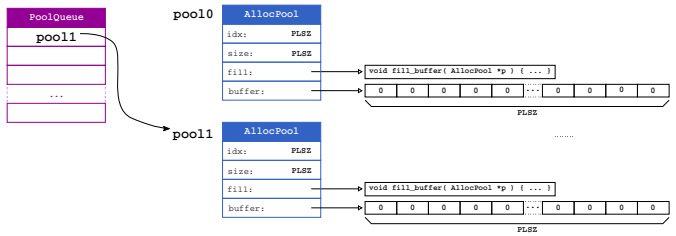
- A set of worker threads, running in parallel with the JavaScript thread (JavaScript is mono-threaded), are in charge of filling buffers of pre-allocated and pre-initialized objects.
- The allocation routine allocates from these buffers.
- Two buffers are used per allocation type so that when one buffer is being filled by an allocation worker the other one is used by the allocator to return fresh objects.

An `AllocPool` holds the *ready* objects, *i.e.,* allocated and pre-initialized objects, that are returned by the allocator. It contains an index `idx` pointing to the next available initialized object, the buffer size, a pointer `fill` to the function to be used for allocating and initializing fresh objects, and the

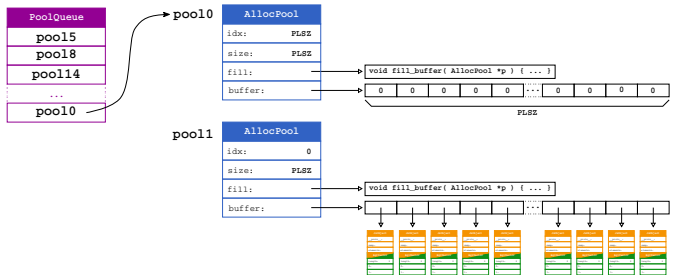
buffer of pre-initialized objects. When empty, `AllocPools` are pushed in the `PoolQueue` from which they are picked in first-in-first-out order by the allocation workers.



(a) Initial worker allocation state.



(b) Intermediate state, the two pools are ready to be filled by the allocation worker.



(c) Ready state, one allocation pool is fully initialized. It can now be used to allocate objects while the other pool is populated by a parallel allocation worker thread.

Fig. 6. Allocation Workers

Figure 6a shows the allocator initial state: the `PoolQueue` is empty and no buffer pool has been allocated yet. For the sake of simplicity we only represent allocation of objects of size 2, *i.e., objects owning two properties*, but the allocator supports heterogeneous objects. Each object kind and each object size must have its own dedicated pair of allocation pools. The first time the allocator is invoked by the client program, the buffers of `pool0` and `pool1` are allocated and filled with null pointers and `pool1` is pushed into the `PoolQueue`. This is depicted by Figure 6b. For this first allocation, after the two pools have been initialized, the allocator invokes the regular sequential allocator to return a fresh object to the main program, as the two `AllocPool` are still empty.

The `pool1` allocation pool will be eventually picked by one allocation worker that will fill its buffer with properly allocated and initialized objects, as shown in Figure 6c. When the buffer is ready, its index `idx` is set to `0`, which marks that it can be used by forthcoming allocations. When this allocation happens, the allocator returns the first fresh object of `pool1` buffer and pushes `pool0` onto the `PoolQueue`. At the time `pool1` buffer is exhausted the allocation buffer of `pool0` is likely to

have been filled already. In that case the two pools are simply swapped. Otherwise, the allocator merely creates objects using regular sequential allocator until `pool0` is ready.

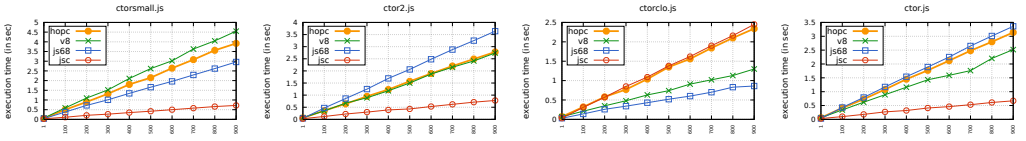
This implementation technique enables to run in parallel object allocations and a significant portion of their initialization. Here is the regular routine in charge of allocating functions:

```
1  obj_t bgl_make_function( obj_t proc, long arity, long ctorsize, obj_t __proto__, obj_t info ) {
2    jsprocedure_t o = GC_MALLOC( sizeof( struct jsprocedure ) );
3
4    o->alloc = jsfunction_alloc; // common initialization
5    o->constrmap = jsfunction_constrmap;
6    o->elements = jsfunction_elements;
7    o->cmap = jsfunction_cmap;
8    o->prototype = _;
9
10   o->__proto__ = __proto__; // instance specific initialization
11   o->procedure = proc;
12   o->arity = arity;
13   o->info = info;
14   o->constrsize = ctorsize;
15   return BGLPROCEDURE( o );
16 }
```

About half the initialization, from line 4 to 8, is common to all created instances. It can then be executed by the parallel worker thread when it allocates the function and stores it in the allocation pool. This enables a faster implementation of `bgl_make_function`.

```
1  obj_t bgl_make_function( obj_t proc, long arity, long ctorsize, obj_t __proto__, obj_t info ) {
2    if( poolfunction.idx < JSFUNCTION_POOLSZ ) {
3      jsprocedure_t o = poolfunction.buffer[ poolfunction.idx ]; // fast allocation
4      poolfunction.buffer[ poolfunction.idx++ ] = 0; // cleanup the buffer to avoid memory leak
5
6      o->__proto__ = __proto__; // instance specific initialization
7      o->procedure = proc;
8      o->arity = arity;
9      o->info = info;
10     o->constrsize = ctorsize;
11     return BGLPROCEDURE( o );
12   } else {
13     // slow path
14   }
15 }
```

The allocation line 3 is faster than the original GC allocation (`GC_MALLOC`) that retrieves fresh pointers from free lists and is close to the performance of a bump allocation.



Three micro-benchmarks (ctorsmall.js⋆, ctor2.js⋆, and ctorclo.js⋆) evaluate the performance of the Hopc parallel allocator. These tests allocate an object in a loop (respectively, a large object, a small object, and a closure) and to prevent the collector to reclaim it instantaneously they store it in a buffer of 10000 elements. Using the parallel allocator, Hopc allocates at a similar pace than the

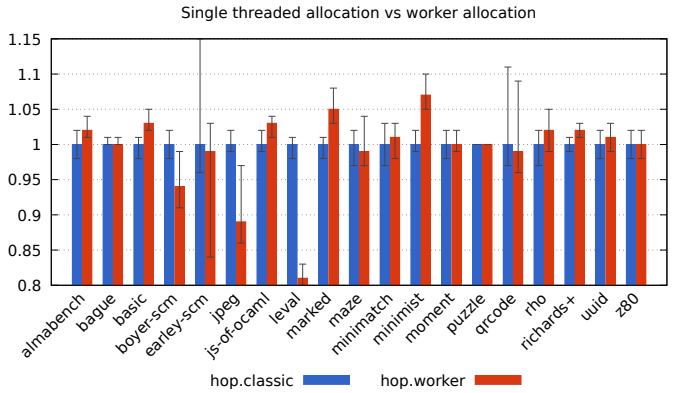


Fig. 7. This experiment compares the performance of Hopc in two configurations. In "classic", objects are allocated using the regular GC allocation function. In "worker", objects are allocated via a dedicated worker thread that runs in parallel with the main JavaScript thread. The vertical axis is the clock wall execution time in seconds relative to single threaded execution, using a linear scale. The measures have been collected on Linux 5.3 x86_64, powered by an Intel Xeon E5-1650. Each test and configuration has been executed 30 times.

other systems. Even when the store buffer is reduced to 10 elements, as in ctor.js*, Hopc speed remains comparable to those of the other systems.

We have measured the impact of the parallel allocator on the whole benchmark suite (Figure 7). The benefit varies from one test to the other. Because it involves multi-threading and thread synchronization it might yield to a small performance slowdown on pathological cases. However, globally, the benefits are more important than the losses.

5.5 Stack Allocation

We have implemented a new Bigloo analysis that aims at replacing heap allocations with stack allocations. This optimization mostly focuses on closures passed to array iterators (`forEach`, `map`, etc.) and on JavaScript `arguments` objects.

The new Bigloo analysis finds heap allocated objects that never escape the lexical block that creates them. These objects are stack allocated. The analysis also keeps track of non-escaping objects stored inside other non-escaping objects so that in addition to flat objects such as *cons* it can also stack allocate compound objects such as Scheme *list*.

The compiler analysis alone is insufficient to optimize objects passed to unknown functions, that is functions that the compiler cannot analyze. Hence, it cannot handle JavaScript frequent patterns such as "`f.apply(mythis, arguments)`", where `arguments` is passed to the unknown `f.apply` method or "`a.forEach(clo)`" where the closure `clo` is passed to the `a.forEach` method. To solve that problem, Hopc uses an optimistic heuristic. It optimistically allocates the object on the stack and it inserts a guard that triggers its reallocation on the heap if the assumption is wrong. Let us illustrate this principle with the implementation of the expression "`a.forEach(x => ...)`" that is compiled into a call to the Scheme function `js-array-maybe-foreach-procedure`:

```
1 (define-inline (js-array-maybe-foreach-procedure this stkproc::procedure thisarg %this cache)
2    (if (js-plain-array? this) ;; is this the builtin forEach JavaScript function that does not capture its argument?
3        (js-array-foreach-procedure this stkproc thisarg %this cache)
4        (let ((heapfun (scheme-stack-procedure->js-function stkproc)))
5           (js-call2 %this (js-get-name/cache this (& "forEach") %this cache)
```

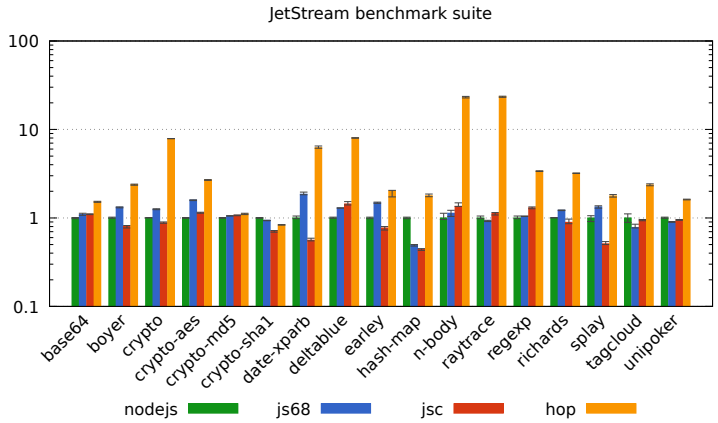
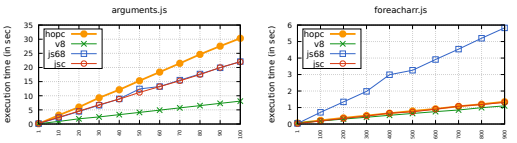


Fig. 8. JetStream performances relative to V8. Lower is better. Logarithmic scale used. Measures collected on Linux 5.3 x86_64, powered by an Intel Xeon E5-1650. Each test and configuration has been executed 30 times.

```
6              this heapfun thisarg))))
7
8  (define (scheme-stack-procedure->js-function::JsFunction stkproc::procedure)
9     (let ((heapproc ($stack-procedure->heap-procedure stkproc)))
10       (js-make-function %this
11          (lambda (self x y z) (heapproc self x y z %this))
12          (js-function-arity 3 0)
13          (js-function-info :name "forEachProc" :len 3)
14          :constrsize 0 :alloc js-object-alloc)))
```

The guard at line 2 succeeds if and only if `this` is an array and its prototype is `Array` and it does not override the `forEach` method. This only involves checking a bit configuration of the `this` header (see Section 4.4). On success the fast path is executed (line 3). Otherwise, the lightweight Scheme stack-allocated closure is copied into the heap (line 9) and a full JavaScript function is allocated (line 10) and the slow path is executed (line 5).

Two micro-benchmarks evaluate the peak performances of `arguments` and `forEach`. As shown by this experiment, Hopc `forEach` performance is on par with those of fast JIT but the stack allocation of `arguments` compensates only partially the discrepancy between Scheme and JavaScript variable arity functions and Hopc is about 2× slower than V8 and 1.35× than Jsc and Js68.



6 PERFORMANCE EVALUATION

This section contains the Hopc performance evaluation. First, in Sections 6.1 and 6.2 we compare its performance with that of JIT compilers. Then, in Sections 6.3 and 6.4 we present an in-depth performance analysis, focusing on Hopc itself.

6.1 Traditional JavaScript Benchmarks

Octane and SunSpider are part of the JetStream2 catalog that contains 64 different tests. It has been extensively used to measure JavaScript implementations performance. All major implementations

have used these tests so intensively, exploiting any peculiarities or even any errors, that they have lost of their value [Meurer 2016; V8 Team 2017]. However, as they constitute a sort of performance milestone, we briefly present how Hopc behaves on these tests but we reserve our in-depth analysis for the new test suite that we present in Section 6.2.

From the JetStream2 set we have excluded tests that are incompatible with Hopc, either because they are implemented as too large files (*e.g.,* Octane typescript and gbemu), because they rely on unsupported features (*e.g.,* JetStream date-format-tofte that uses `eval` to access the lexical environment or the WSL suite that tests wasm implementations). Figure 8 shows the performance relative to V8.

For many tests, Hopc performance is in the range 1×..2× when compared to V8. We read as a good result considering the energy spent by all JIT compilers to optimize these very tests. However, a minority of tests reveal performance issues.

Unsurprisingly Hopc is inefficient on floating-point number tests (crypto, n-body, and raytrace) where most of the execution is spent allocating boxed numbers, reclaiming boxed numbers, and checking the types of generic arithmetic operators that never fail. Hopc is also inefficient on regexp because of its lack of regular expressions optimizations and slow backend. Hopc encodes non-ascii strings into UTF-8 format, which penalizes the performance of crypto-aes since it uses non-ascii strings extensively, demanding 16-bit encoding for performance. The test date-xparb heavily relies on JavaScript `eval`. Hopc barely supports it by dynamically compiling JavaScript into Scheme and then executing the obtained Scheme expression using an unmodified Bigloo interpreter. This is obviously one of the main limitations of the full AoT approach, as it cannot deliver good performance for programs relying on `eval`. Efficency on such tests requires either a JIT compilers or a hybrid AoT and JIT approach.

The `deltablue` test shows weak Hopc performance. It is the transcription of a Smalltalk test that mimics Smalltalk-style object-oriented programming. As such, it uses method invocation extensively. Most of these invoked methods are small and some are even empty. JIT compilers are able to inline or remove entirely these calls. Hopc does not, as this is difficult to achieve in an AoT setting. As of today, this is probably the second most important limit of the AoT approach. Profile-guided optimizations could help mitigating this problem and it is a direction for future work.

6.2 New Test Suite

Figure 9 presents the performance comparison of JIT compilers and Hopc on the test suite presented in Section 3. Before we analyze each test individually we make four global observations.

Observation 1: The performance diversity is important, even among JIT compilers. For instance, for the jpeg.js test, the slowest JIT compiler is 9× slower than the fastest. We observe a 2× or more ratio between the fastest and slowest JIT compilers for about half of the tests.

Observation 2: Which compiler performs best or worst varies from one benchmark to the other. For instance Js68 is the slowest JIT for earley-scm.js, maze.js, and z80.js but it is the fastest for basic.js and qrcode.js. Jsc is the slowest JIT for bague.js but it is the fastest for boyer-scm.js, earley-scm.js, leval.js, rho.js, uuid.js, and z80.js.

Observation 3: If JIT compilers suffer from a slow start with a ramp up period, it is unnoticeable when the executions last half a second or more. These JIT compilers are so efficient that all the curves of their execution times appear to be continuous functions.

Observation 4: Apart from almabench.js and moment.js, Hopc delivers performance comparable to that of JIT compilers, among the fastest on some tests, among the slowest on others.

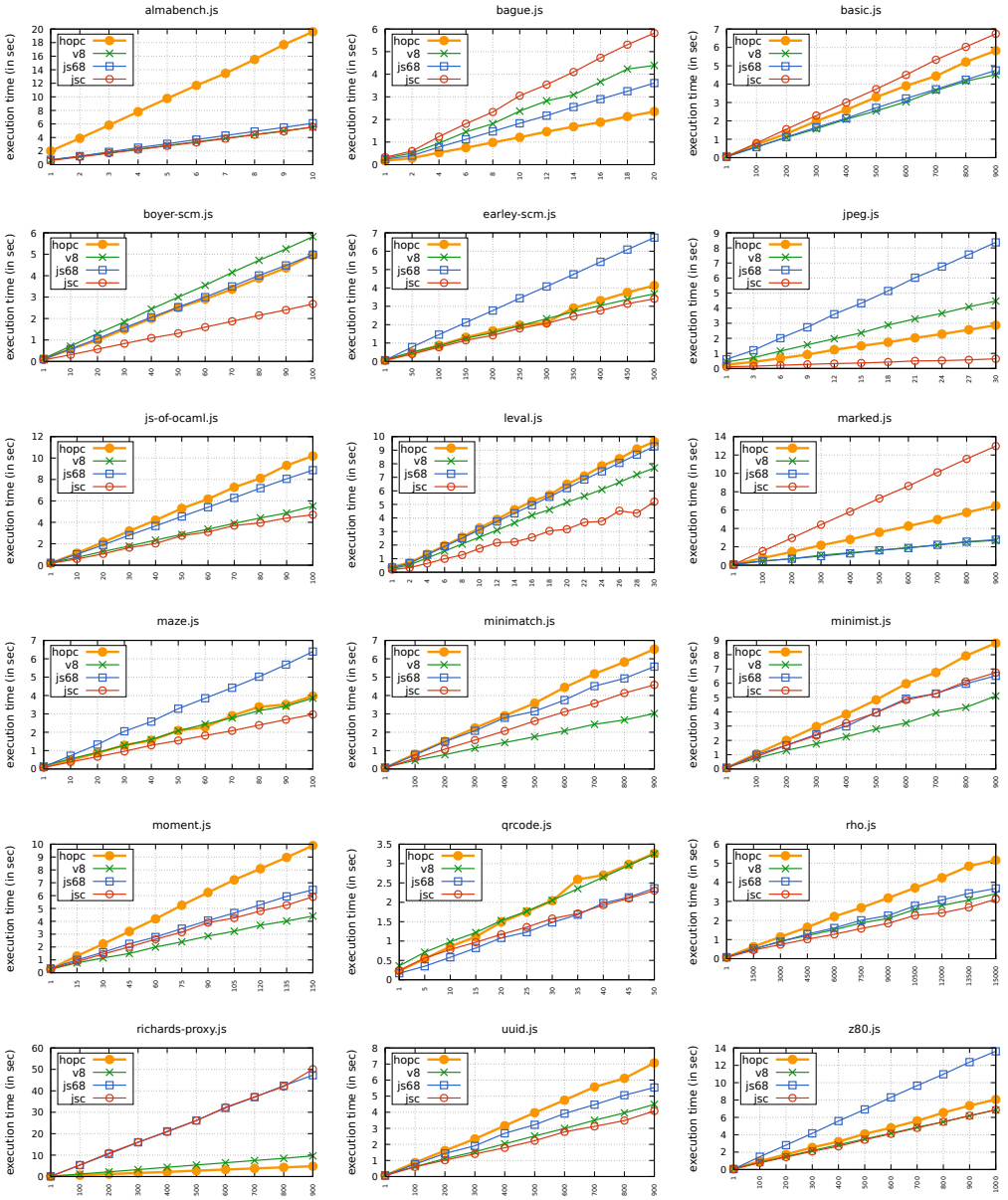


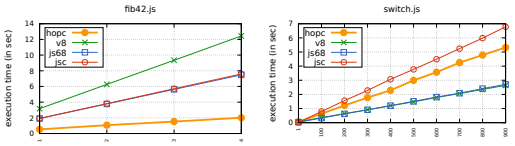
Fig. 9. JIT performances (V8 (6.8.275.32), Jsc (4.0), and Js68 (C68)) vs aot performances (hop). The vertical axis are the clock wall execution times. Smaller is then better. The horizontal axis are the numbers of iterations of the benchmark. Measures collected on Linux 5.3 x86_64, powered by an Intel Xeon E5-1650.

6.3 Performance Analysis

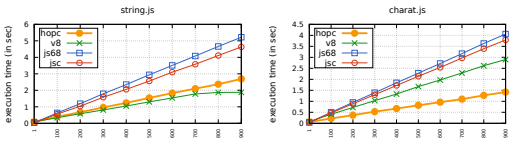
In this section we conduct a per-test analysis using some of our micro-benchmarks.

almabench.js is a floating point intensive program. Hopc is so penalized (about 3× slower than V8) by its lack of floating point optimization that it is not competitive. The allocation problem is

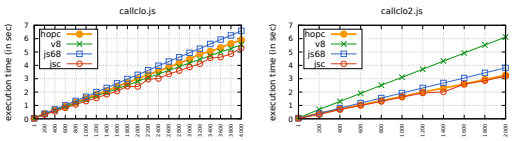
so obvious that we did not analyze any further this weak result. Hopc is not ready for floating programs yet!



bague.js performance is dominated by fixed integer arithmetic, array accesses, and simple recursive functions. The Hopc type and range analyses are powerful enough to infer small integer types. Hopc also fully benefits from its efficient direct recursive calls, and from its efficient integer dispatch that are visible on the fib42.js⋆ and switch.js⋆ micro-benchmarks.



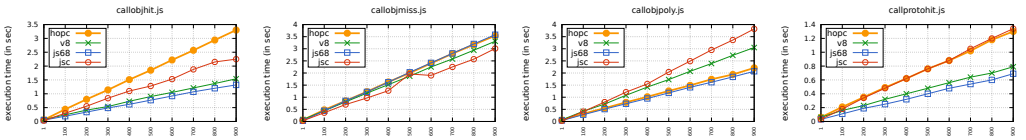
basic.js spends more than 25% of the Hopc execution time in a function that parses tokens. This function splits source lines and compares token characters one by one. The allocation of short lived tokens penalizes Hopc (see Section 5.4) but it catches up on the other parts of the program because its optimistic compilation succeeds in correctly guessing function types, which enables the generated code to use efficient string manipulation functions whose good performance is visible in the string.js⋆, which concatenates strings and searches for a pattern with `indexOf`, and charat.js⋆, which accesses string characters.



boyer-scm.js exercises intensively direct property access. This test shows that in spite of the extra memory read of property accesses (Section 5.2), Hopc is competitive for accessing and setting object properties. In this test, 100% of object properties are inlined (see Section 4.4, Figure 3), which contributes to the good overall performances.

This benchmark uses closure invocation as they are intensively used in the Scheme high level operators `map` and `for-each`. The tests callclo.js⋆ and callclo2.js⋆ show that Hopc can generate efficient code and efficient closures when its analysis lets it eliminate JavaScript function constructions (see Section 4.6). This benefits this benchmark.

This test also uses methods invocation extensively. Four micro-benchmarks evaluate the speed of method invocation in four contexts: *i)* the method is located inside the object and hits the inline cache, *ii)* the method is in the object but reassigned so that each call misses the cache, *iii)* the method is in the object but the objects are polymorphic, and *iv)* the method is found in the prototype chain.

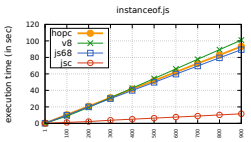


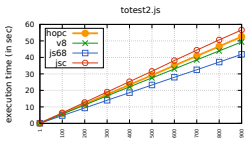
Hopc does not match JIT performance on callobjhit.js⋆ and callprotohit.js⋆ because it does not inline the small method used in these tests. Closures inlining is difficult for AoT compilers and probably is one significant advantage of JIT compilers. The good performance obtained on callobjmiss.js⋆ and callobjpoly.js⋆ indicates that Hopc supports polymorphism as efficiently as a JIT.

earley-scm.js tests intensively recursive functions used for traversing lists. The termination test is generally implemented as below:

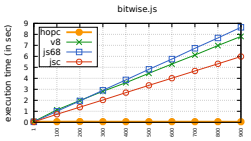
```
function pairp(o) { return o instanceof pair; }
```

```
const loop1 = function loop1(l1, l2) { if(pairp(l1)) { ... } else { ... } }
```





The condiontal expression `pairp(l1)` would be a mundane expression in any other language, but turns out to be complex to implement efficiently for JavaScript. Besides inlining the `pairp` call, something all systems probably do, the compiler faces the problem of efficiently compiling the Boolean test as many JavaScript values cause `if` to take the else branch (the empty string, the number `0`, the undefined value, etc.). When the Hopc type analyses prove that a test expression evaluates to a Boolean value, the test is compiled as a mere comparison. totest2.js* shows that Hopc does this as efficiently as the JIT compilers. The class predicate `instanceof` also has to be implemented efficiently but the prototype-based nature of JavaScript makes this difficult. The test instanceof.js* evaluates this aspect. Apart from Jsc, which is lightning fast, Hopc performs as well as the other JIT compilers.

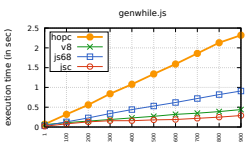


jpeg.js uses bitwise operations (bitwise.js*) and array read/write extensively. The Hopc type and array range analyses are sufficiently powerful to let the compiler produce efficient code that uses fixnum arithmetic.

leval.js exercises closure invocations (see callclo.js* and callclo2.js*) and object allocations (see Section 5.4). It allocates many short-lived objects which penalizes Hopc, as it uses a mark&sweep garbage collector. More than half of the Hopc execution time is spent in the garbage collector.

js-of-ocaml.js is an allocation intensive program. It keeps allocating arrays of three elements for implementing OCaml activation frames. More than 50% of the overall hop execution is spent in the garbage collector.

marked.js mostly regular expressions for search and replace patterns. The Hopc implementation of some of these functions has not been polished enough yet to let it compete with the fastest JIT compilers.



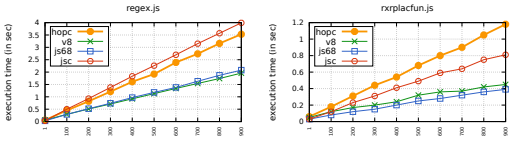
maze.js is among the very few test to use ES6 generators (for implementing a random generator). Hopc compiles them using a partial CPS transform. The benchmark genwhile.js* tests this feature and shows that here again, Hopc is penalized by its non-compacting collector (V8, Jsc, and Js68 use compacting collectors). Generators allocate continuations that generally have extremely brief lifetimes. Hopc allocates them in the heap where they stay, awaiting the next sweep phase. Still, a good performance is still obtained for maze.js because the Hopc closure optimizations are as efficient as those used by the JIT compilers (see Section 4.6 and see ctorclo.js*).

minimatch.js suffers from the slow implementation of hop exceptions. A JavaScript `try` statement has to be compiled into a `setjmp` because the JavaScript function calls use the C stack. As `setjmp` only restores the values of the current function arguments and not the values of temporary variables, compiling `try` requires Hopc to cut functions into pieces, which slows down the control flow and forces some JavaScript variables to be accessed via heap cells. The optimization described in Section 5.3 that removes `setjmp`/`longjmp` for `return`, `break`, and `continue` does not apply to `try`/`catch`.

minimist.js execution is dominated by array constructions obtained by splitting strings. Most of these arrays have a very brief lifetime. This is why Hopc does not perform so well on this test.

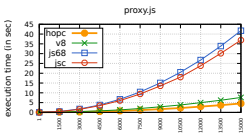
About 25% of the Hopc execution time is spent collecting dead objects. This prevents Hopc to compete with systems equipped with faster garbage collectors.

moment.js uses regular expressions for parsing dates. Hopc does not handle them very efficiently (in particular see rxrplacfun.js★). In addition, the test uses dates extensively. They have poor performance because Hopc uses Posix dates and `localtime`, which rely on `tzset`, which itself manipulates the user's environment variables and acquires a global lock to preventing concurrent access.



qrcode.js is the only test of our suite that uses objects as dictionaries. These objects are not implemented efficiently with hidden classes because keys indexing properties are dynamic. They are better handled with hash tables that most systems, including Hopc, implement. This test shows that Hopc can switch from one object representation to another as efficiently as JIT compilers.

richards+.js allocates and uses proxy objects extensively. This test shows that Hopc can efficiently cope with the new ES6 features and that it can deploy efficient strategies for supplanting simple inline caches when accessing properties whose names are not known statically. The technique used by Hopc has been described in a previous publication [Serrano and Findler 2020].



rho.js uses arrays extensively. More precisely, it frequently pushes new values into existing arrays. This operation requires techniques to efficiently resize existing arrays. The test shows that Hopc handles these dynamic arrays reasonably efficiently.

As presented in Section 5.5, the Hopc `arguments` implementation is not competitive with JIT compilers that support it natively. In this benchmark the situation is even worse because the Hopc static analysis is not powerful enough to allocate `arguments` on the heap.

uuid.js extensively tests 32bit operators, string manipulations, array constructions, and exceptions (used to detect tests that *must* fail). Uuid.js suffers like minimatch.js because of the slow implementation of `try/catch` but it catches up with its type inference and optimistic compilation.

Uuid.js performance depends on functions such as this one:

```
function str2binl(str) {
  var bin = Array(); var m = (1 << chrsz) - 1;
  for(var i = 0; i < str.length * chrsz; i += chrsz)
    bin[i>>5] |= (str.charCodeAt(i / chrsz) & m) << (i%32);
  return bin;
}
```

Taking advantage of the 32-bit integer specification of JavaScript bit-wise operators, local reasoning is enough to generate efficient code for bit shifts and logical operations, and Hopc handles them more efficiently than all other systems (see bitwise.js★). Constant propagation enables Hopc to compile `i/chrsz` into a mere right bit shift as the variable `chrsz` happens to be the constant `8`. The problem of compiling `str` accesses efficiently remains, however. If the compiler is not able to prove that `str` is a string, the expression `str.length` is not known to be an integer and thus `charCodeAt` is not known to return an integer. Consequently, the `for` loop will be cluttered with tests and generic arithmetic operators.

In the test, `str2binl` is used in such a way that Hopc cannot prove that `str` is always a string, but applying syntactic rules [Serrano 2018] to the function definition it establishes that it could generate a much better code *if* `str` was a string. Accordingly, Hopc generates two versions of the

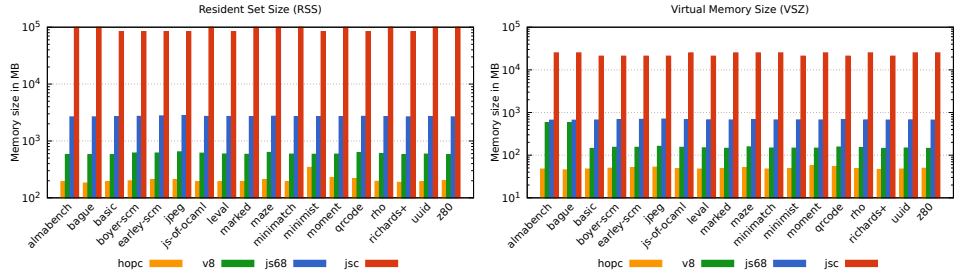


Fig. 10. Memory footprints measured by the Unix ps tool. A logarithmic scale is used. Lower is better.

`str2binl` and, at call sites for which `str` type is unknown it uses a dynamic check and dispatches. For uuid.js, it always selects the efficient version.

z80.js is handled efficiently by Hopc whose good performance comes from its type analyses and optimistic compilation that generate fast code for z80 instruction decoding and evaluation. Hopc also benefits from its good support for integer operations.

6.4 Memory

We have compared the memory usage of the JIT compilers and Hopc. Figure 10 shows the memory usage reported by the Unix "ps" tool. It shows how much RAM memory is allocated per program (RSS) and the virtual memory size (VSZ), i.e., the overall memory use including swap and program use. This experiment shows that AoT can fulfill the promise of using less memory. Program sizes and the dynamic memory are drastically reduced; Hopc uses at least 3× less memory than V8, the most memory-efficient JIT.

7 CONCLUSION

Fundamentally, it is difficult to conclude anything more than AoT compilers for JavaScript are viable based on the evidence presented in this work. Could they compete or not in term of pure speed is still unanswered. The truth hinges on two questions. Is it the several decades of tuning to the benchmarks that gives JIT compilers the edge over Hopc on JetStream or is it their inherent JIT nature? Is it possible to build an AoT compiler that optimizes the entire JavaScript language and that delivers performance globally comparable to those of fast JIT compilers? Our own experience with compilers suggests is that it is inevitable that the performance of the compiler grows more and more tuned to standard benchmarks; accordingly we put more weight on the new benchmarks that were developed for this paper as revealing the truth. The second question, remains open. See this paper, then, as a call to arms: let us bring an answer to that question!

Hopc is publicly available and its architecture makes it easy to extend. New passes, implemented in Scheme or JavaScript, can be easily added because the compiler can export its abstract syntax after any compilation stage and resume the compilation after the invocation of the external pass. It can be used as a testbed for experimenting with new analyses and new optimizations for dynamic languages in general, and for JavaScript in particular.

☞ **url:** http://hop.inria.fr

ACKNOWLEDGMENTS

Toute ma gratitude et mes remerciements à Robby Findler pour son aide et ses conseils précieux.

A PEAK PERFORMANCE

We have gathered execution times of JetStream and the new benchmark suite 3 with many JavaScript implementations. They are presented in Figures 11 and 12. All measures presented have been collected on Linux 5.3 x86_64, powered by Intel a Xeon E5-1650 processor. The tested systems are : V8 (6.8.275.32), Jsc (4.0), Js68 (C68), Graal (19.1.1), QuickJS$^{\varepsilon}$ (2020-11-08), JerryScript$^{\varepsilon}$ (2.4.0), Rhino (1.7.7.2). Interpreters are distinguished with the $^{\varepsilon}$ mark.

benchmark	hop	v8	js68	jsc	qjs	V8$^{\text{jitless}}$	graal	jerry	rhino
base64	14.28 0.7%	9.38 0.8%	10.18 1.2%	10.36 0.5%	210.25 0.1%	104.36 0.4%	36.80 0.1%	447.04 0.0%	218.93 0.5%
boyer	15.66 0.6%	6.59 1.0%	8.71 0.5%	5.27 1.5%	162.27 1.0%	62.12 0.3%	19.58 0.5%	_	88.06 3.0%
crypto	57.65 0.2%	7.33 0.3%	9.19 0.7%	6.46 0.8%	247.69 0.2%	261.15 0.2%	17.44 1.6%	906.44 0.0%	251.16 8.8%
crypto-aes	14.54 0.5%	5.40 0.5%	8.56 0.4%	6.15 0.6%	144.11 0.1%	122.98 0.3%	36.80 0.7%	290.16 0.0%	76.76 1.6%
crypto-md5	3.75 1.0%	3.38 0.4%	3.56 0.3%	3.61 0.6%	86.50 0.0%	100.47 0.1%	23.91 2.3%	398.82 0.0%	90.80 1.6%
crypto-sha1	8.81 0.5%	10.60 0.4%	9.93 0.2%	7.50 1.1%	257.42 0.1%	353.26 0.1%	49.36 0.8%	1228.55 0.0%	184.56 1.1%
date-xparb	10.80 1.3%	1.74 1.2%	3.22 1.5%	0.98 1.5%	9.28 0.1%	3.91 0.9%	3.19 1.5%	14.41 0.0%	_
deltablue	33.56 0.3%	4.18 0.8%	5.41 0.2%	6.00 1.8%	364.95 0.2%	219.78 0.4%	18.95 0.6%	1068.12 0.0%	295.75 2.7%
earley	13.33 4.2%	6.60 0.9%	9.76 0.7%	5.06 1.7%	_	50.31 0.2%	43.95 1.3%	_	_
hash-map	20.88 2.0%	11.72 0.8%	5.70 1.1%	5.15 0.9%	135.03 0.1%	84.03 0.3%	5.36 0.6%	_	128.27 1.1%
n-body	5.31 0.8%	0.23 3.7%	0.26 3.3%	0.31 2.3%	5.73 0.2%	7.35 0.5%	1.99 3.7%	24.78 0.0%	7.28 1.3%
raytrace	29.77 0.8%	1.27 1.6%	1.17 0.9%	1.42 1.7%	84.57 0.3%	28.14 0.4%	5.23 3.0%	198.43 0.0%	64.08 3.3%
regexp	22.56 0.4%	6.69 1.3%	6.98 0.3%	8.69 1.2%	183.79 0.7%	22.84 1.1%	85.85 0.7%	_	118.09 5.6%
richards	24.69 0.2%	7.74 0.2%	9.42 0.4%	6.90 2.5%	293.62 0.1%	205.53 0.3%	16.26 1.4%	1018.72 0.0%	198.24 1.1%
splay	10.16 1.4%	5.75 3.3%	7.66 1.2%	2.96 1.6%	78.29 0.8%	22.50 0.4%	27.34 3.1%	_	16.89 1.0%
tagcloud	19.59 1.1%	8.31 5.8%	6.45 2.2%	7.88 1.0%	149.26 0.2%	13.04 0.6%	22.58 0.1%	_	56.13 3.3%
unipoker	11.40 0.5%	7.07 0.6%	6.33 0.6%	6.69 0.6%	62.65 0.3%	37.86 0.4%	19.46 3.2%	143.78 0.0%	_

Fig. 11. Peak performance JetStream JavaScript programs. The table reports the mean of 30 execution times (clock wall time) and in a smaller font, the deviation of each test.

benchmark	hop	v8	js68	jsc	qjs	V8$^{\text{jitless}}$	graal	jerry	rhino
almabench	19.42 0.5%	5.44 0.3%	6.09 0.3%	5.42 0.1%	71.39 0.1%	59.72 0.4%	9.53 0.4%	159.09 0.0%	47.37 4.0%
bague	4.63 0.2%	8.78 2.8%	7.13 0.3%	12.16 1.8%	313.54 0.1%	258.39 0.3%	37.13 0.3%	931.19 0.0%	126.04 1.3%
basic	6.52 1.0%	5.04 1.6%	5.10 1.9%	7.63 1.0%	116.91 0.0%	60.52 0.3%	14.92 5.5%	_	_
boyer-scm	2.00 2.1%	2.41 0.9%	2.03 0.6%	1.06 1.3%	38.11 1.7%	21.00 0.4%	7.61 0.9%	_	_
earley-scm	4.01 6.8%	3.74 1.2%	6.76 0.5%	3.45 1.5%	_	26.21 1.7%	_	_	_
jpeg	2.84 1.5%	4.50 1.0%	8.22 1.0%	0.66 3.1%	13.92 0.1%	7.38 0.6%	27.33 0.6%	_	_
js-of-ocaml	10.29 1.0%	5.46 0.6%	8.91 0.3%	4.91 3.9%	54.26 0.1%	40.01 0.5%	17.00 1.8%	175.52 0.0%	59.88 9.0%
leval	12.90 0.7%	10.21 0.4%	12.35 0.2%	6.65 4.2%	202.16 0.4%	104.18 0.2%	19.08 1.1%	542.86 0.0%	_
marked	7.19 1.3%	2.92 1.9%	3.07 1.7%	14.42 0.3%	86.61 0.1%	11.23 0.4%	18.87 4.3%	_	_
maze	7.86 2.1%	7.46 0.7%	12.34 0.9%	5.87 0.6%	_	28.63 0.3%	15.17 0.7%	_	_
minimatch	7.51 1.2%	3.26 2.1%	6.48 1.5%	5.04 1.1%	51.84 0.5%	21.48 0.9%	11.73 1.8%	109.69 0.0%	_
minimist	9.86 1.2%	5.16 1.7%	7.48 4.3%	7.58 0.9%	40.89 0.2%	15.63 0.7%	23.33 1.4%	53.49 0.0%	_
moment	6.90 0.7%	3.04 2.6%	4.42 1.3%	4.11 1.5%	15.45 0.2%	7.54 0.4%	30.66 1.0%	_	19.66 2.6%
puzzle	11.44 0.1%	9.46 0.2%	9.18 0.4%	8.17 0.4%	224.29 0.1%	175.51 0.2%	53.18 1.6%	489.69 0.0%	86.48 1.1%
qrcode	3.35 3.9%	3.18 0.7%	2.34 1.2%	2.29 2.7%	16.52 0.2%	8.98 0.3%	9.82 1.8%	_	17.53 3.3%
rho	7.11 1.4%	4.34 2.2%	4.89 1.9%	3.96 1.7%	22.25 0.4%	12.33 0.5%	33.35 0.9%	54.97 0.0%	30.02 2.1%
richards+	5.18 1.0%	10.61 1.0%	52.57 0.6%	53.56 1.7%	25.29 0.5%	18.41 0.3%	15.57 3.8%	51.88 0.0%	_
uuid	7.93 1.9%	4.86 1.4%	6.21 1.4%	4.41 1.3%	40.55 0.1%	34.68 0.2%	53.32 0.4%	122.22 0.0%	_
z80	8.01 1.2%	6.84 0.3%	13.68 0.3%	6.83 1.2%	82.21 0.1%	12.31 0.5%	30.41 3.4%	_	_

Fig. 12. Peak performance of the 18 JavaScript programs. The table reports the mean of 30 execution times (clock wall time) and in a smaller font, the deviation of each test.

REFERENCES

W. Ahn et al. 2014. Improving javascript performance by deconstructing the type system. In *PLDI 2014 - Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI))*. Association for Computing Machinery, Edinburgh, UK, 496–507. https://doi.org/10.1145/2594291.2594332 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2014 ; Conference date: 09-06-2014 Through 11-06-2014.

C. Anderson, P. Giannini, and S. Drossopoulou. 2005. Towards Type Inference for Javascript. In *Proceedings of the 19th European Conference on Object-Oriented Programming* (Glasgow, UK) *(ECOOP'05)*. Springer-Verlag, Heidelberg. https://doi.org/10.1007/11531142_19

Apple. 2018. WebKit. https://webkit.org/.

Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static Analysis for ECMAScript String Manipulation Programs. *Applied Sciences* 10, 10 (May 2020), 3525. https://doi.org/10.3390/app10103525

R. Artoul. 2015. Javascript Hidden Classes and Inline Caching in V8. http://richardartoul.github.io/jekyll/update/2015/04/26/hidden-classes.html.

Thomas Ball, Peli de Halleux, and Michał Moskal. 2019. Static TypeScript: An Implementation of a Static Compiler for the TypeScript Language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes* (Athens, Greece) *(MPLR 2019)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/3357390.3361032

Spenser Bauman, Carl Friedrich Bolz, Robert Hirschfeld, Vasily Kirilichev, Tobias Pape, Jeremy G. Siek, and Sam Tobin-Hochstadt. 2015. Pycket: A Tracing JIT for a Functional Language. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 22–34. https://doi.org/10.1145/2784731.2784740

H.J. Boehm and M. Weiser. 1988. Garbage Collection in an Uncooperative Environment. *Software — Practice and Experience* 18, 9 (Sept. 1988), 807–820. https://doi.org/10.1002/spe.4380180902

Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Towards a Verified Range Analysis for JavaScript JITs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 135–150. https://doi.org/10.1145/3385412.3385968

C. Chambers and D. Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language. In *Conference Proceedings on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA. https://doi.org/10.1145/73141.74831

C. Chambers, D. Ungar, and E. Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications* (New Orleans, Louisiana, USA) *(OOPSLA '89)*. ACM, USA, 49–70. https://doi.org/10.1145/74878.74884

S. Chandra et al. 2016. Type Inference for Static Compilation of JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. ACM, New York, NY, USA. https://doi.org/10.1145/3022671.2984017

M. Chevalier-Boisvert and M. Feeley. 2015. Simple and Effective Type Check Removal through Lazy Basic Block Versioning. In *29th European Conference on Object-Oriented Programming, ECOOP 2015* (Prague, Czech Republic). https://doi.org/10.4230/LIPIcs.ECOOP.2015.101

M. Chevalier-Boisvert and M. Feeley. 2016. Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. In *30th European Conference on Object-Oriented Programming* (Rome, Italy). https://doi.org/10.4230/LIPIcs.ECOOP.2016.7

Jiho Choi, Thomas Shull, and Josep Torrellas. 2019. Reusable Inline Caching for JavaScript Performance. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 889–901. https://doi.org/10.1145/3314221.3314587

W. Choi, S. Chandra, G. Necula, and L. Sen. 2015. SJS: A Type System for JavaScript with Fixed Object Layout. In *Static Analysis - 22nd International Symposium, SAS'15*. Saint-Malo, France, 181–198. https://doi.org/10.1007/978-3-662-48288-9_11

D. Clifford, H. Payer, M. Stanton, and B. Titzer. 2015. Memento Mori: Dynamic Allocation-site-based Optimizations. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. New York, NY, USA. https://doi.org/10.1145/2887746.2754181

G. Dot et al. 2017. Removing Checks in Dynamically Typed Languages through Efficient Profiling. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization* (Austin, USA) *(CGO '17)*. IEEE Press, 257–268. https://doi.org/10.5555/3049832.3049860

ECMA International. 2015. *Standard ECMA-262 - ECMAScript Language Specification* (6.0 ed.).

ECMA International. 2018. *ECMAScript 2018 Language Specification* (9.0 ed.). https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

David Eder. 2020. a simple BASIC interpreter. http://eder.us/projects/jbasic/

D. Flanagan. 2002. *JavaScript – The definitive guide (fourth edition)*. O'Reilly & Associates, USA.

A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. 2009a. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*. https://doi.org/10.1145/1542476.1542528

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. 2009b. Trace-Based Just-in-Time Type Specialization for Dynamic Languages. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

Google. 2018. V8 JavaScript Engine. http://developers.google.com/v8

Google. 2019. JIT-less V8. https://v8.dev/blog/jitless.

A. Guha, C. Saftoiu, and S. Krishnamurthi. 2010. The essence of JavaScript. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'10)*. https://doi.org/10.5555/1883978.1883988

Andreas Haas et al. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

B. Hackett and S-Y. Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China). New York, NY, USA. https://doi.org/10.1145/2345156.2254094

James Halliday. 2020. Minimist: parse argument options. https://github.com/substack/minimist

Zoltan Herczeg. 2015. Performance comparison of regular expression engines. https://zherczeg.github.io/sljit/regex_perf.html

U. Hölzle, C. Chambers, and D. Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. UK, 21–38. https://doi.org/10.1.1.126.7745

Adobe Systems Incorporated. 2020. jpeg.js, A pure javascript JPEG encoder and decoder for node.js. https://github.com/eugeneware/jpeg-js

Isaacs. 2016. https://github.com/isaacs/minimatch

Christopher Jeffrey et al. 2020. Marked. https://github.com/markedjs/marked

S. Jensen, A. Møller, and P. Thiemann. 2009a. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)* (Los Angeles, CA). Springer-Verlag, Berlin, Heidelberg, 238–255. https://doi.org/10.1007/978-3-642-03237-0_17

Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009b. Type Analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium (SAS) (LNCS, Vol. 5673)*. Springer-Verlag. https://doi.org/10.1007/978-3-642-03237-0_17

S H. Jensen, A. Møller, and P. Thiemann. 2009c. Type Analysis for JavaScript. In *Proceedings of the 16th International Symposium on Static Analysis* (Los Angeles, CA) *(SAS '09)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03237-0_17

Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A Static Analysis Platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 121–132. https://doi.org/10.1145/2635868.2635904

Arase Kazuhiko. 2009. QRcode.js. http://www.d-project.com/

R. Kelsey, W. Clinger, and J. Rees. 1998. The Revised(5) Report on the Algorithmic Language Scheme. *Higher-Order and Symbolic Computation* 11, 1 (Sept. 1998).

Y. Ko, H. Lee, J. Dolby, and S. Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 541–551. https://doi.org/10.1109/ASE.2015.28

B. S. Lerner, J. Politz, J.G., A. Guha, and S. Krishnamurthi. 2013. TeJaS: Retrofitting Type Systems for JavaScript. In *Proceedings of the 9th Symposium on Dynamic Languages* (Indianapolis, Indiana, USA) *(DLS '13)*. ACM, NY, USA. https://doi.org/10.1145/2578856.2508170

F. Logozzo and H. Venter. 2010. RATA: Rapid Atomic Type Analysis by Abstract Interpretation - Application to JavaScript Optimization. In *Compiler Construction, 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. https://doi.org/10.1007/978-3-642-11970-5_5

Florian Loitsch. 2005. Javascript to Scheme Compilation. In *Proceedings of the Sixth Workshop on Scheme and Functional Programming*. 101–116.

F. Loitsch and M. Serrano. 2008. *Trends in Functional Programming*. Vol. 8. Seton Hall University, Intellect Bristol, UK/Chicago, USA, Chapter Hop Client-Side Compilation, 141–158.

MDN. 2019. SpiderMonkey Internals. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Internals

B. Meurer. 2016. The truth about traditional JavaScript benchmarks. https://benediktmeurer.de/2016/12/16/the-truth-about-traditional-javascript-benchmarks.

Moment.com. 2020. Moment.js. https://momentjs.com/

Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. 2020. Value and Allocation Sensitivity in Static Python Analyses. In *Proceedings of the 9th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis* (London, UK) *(SOAP 2020)*. Association for Computing Machinery, New York, NY, USA, 8–13. https://doi.org/10.1145/3394451.3397205

Mozilla. 2020. SpiderMonkey: The Mozilla JavaScript runtime. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems* (Washington, DC, USA) *(ASPLOS '09)*. ACM, New York, NY, USA, 265–276. https://doi.org/10.1145/1508244.1508275

Benjamin Barslev Nielsen and Anders Møller. 2020. Value Partitioning: A Lightweight Approach to Relational Static Analysis for JavaScript. In *Proc. 34th European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2020.16

C. Park and S. Ryu. 2015. Scalable and Precise Static Analysis of JavaScript Applications via Loop-Sensitivity. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, Prague, Czech Republic*. https://doi.org/10.1145/3093334.2989228

Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-Driven Incremental Inline Substitution Algorithm for Just-in-Time Compilers. In *Proceedings of the 2019 International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 164–179. https://doi.org/10.5555/3314872.3314893

M. Qunaibit et al. 2018. Accelerating Dynamically-Typed Languages on Heterogeneous Platforms Using Guards Optimization. In *Proceedings of 32th the European Conference on Object-Oriented Programming (ECOOP'18)* (Amsterdam, NL). https://doi.org/10.4230/LIPIcs.ECOOP.2018.16

Brianna M. Ren and Jeffrey S. Foster. 2016. Just-in-Time Static Type Checking for Dynamic Languages. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 462–476. https://doi.org/10.1145/2908080.2908127

V. Saint-Amour and S-Y Guo. 2015. Optimization Coaching for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015* (Prague, Czech Republic). https://doi.org/10.4230/LIPIcs.ECOOP.2015.271

Manuel Serrano. 1992. Bigloo, a Practical Scheme Compiler. http://www-sop.inria.fr/indes/fp/Bigloo/

M. Serrano. 2018. JavaScript AOT Compilation. In *14th Dynamic Language Symposium (DLS)*. Boston, USA. https://doi.org/10.1145/3276945.3276950

M. Serrano and M. Feeley. 2019. Property Caches Revisited. In *Proceedings of the 28th Compiler Construction Conference (CC'19)*. Washington, USA. https://doi.org/10.1145/3302516.3307344

M. Serrano and R. Findler. 2020. Dynamic Property Caches, a Step towards Faster JavaScripts Proxy Objects. In *Proceedings of the 29th Compiler Construction Conference (CC'20)*. San Dieo, USA. https://doi.org/10.1145/3377555.3377888

O. Shivers. 1988. Control Flow Analysis in Scheme. In *Proceedings of the SIGPLAN '88* ACM Sigplan *Int'l Conference on Programming Language Design and Implementation (PLDI)*. Atlanta, Georgia. https://doi.org/10.1145/960116.54007

S. Strickland et al. 2012. Chaperones and Impersonators: Run-time Support for Reasonable Interposition. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*. Arizona, USA, 943–962. https://doi.org/10.1145/2384616.2384685

UUID 2020. uuid.js, For the creation of RFC4122 UUIDs. https://www.npmjs.com/package/uuid

V8 Team. 2017. Retiring Octane. https://v8.dev/blog/retiring-octane.

Jérôme Vouillon et al. 2020. Js_of_ocaml. https://ocsigen.org/js_of_ocaml

April W. Wade, Prasad A. Kulkarni, and Michael R. Jantz. 2017. AOT vs. JIT: Impact of Profile Data on Code Quality. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Barcelona, Spain) *(LCTES 2017)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3078633.3081037

Wikipedia. 2021. W∧X. https://en.wikipedia.org/wiki/W%5EX.

Christian Wimmer et al. 2019. Initialize Once, Start Fast: Application Initialization at Build Time. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 184 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360610

T. Würthinger et al. 2017. Practical Partial Evaluation for High-Performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381