A Snapshot of the Performance of Wasm Backends for Managed Languages

Manuel Serrano
Inria/Université Côte d'Azur
Sophia Antipolis, France
Manuel.Serrano@inria.fr

Robert Bruce Findler
Northwestern University
Evanston, IL, USA
robby@northwestern.edu

Abstract

WebAssembly (Wasm) has been extended to support features such as garbage collection, references, exceptions, and tail calls that facilitate compilation of managed languages. In this paper, we capture a snapshot of the performance of languages that use these new capabilities from two perspectives. First, we present a language-by-language performance comparison of six managed language implementations on Wasm to the performance to their native implementations. Second, we focus on the implementation of the Bigloo Scheme compiler and explore the impact of different choices for compiling specific aspects of the language. Our findings suggest that Wasm has become a promising compilation target for most managed languages, but that its performance still falls short of that achieved by native code. Our results also show that the quality of the Wasm implementations vary, with the best ones being, on average, about 1.4× slower than the native backend and the worst ones seeing average slowdowns of more than 8× with some tests even failing to execute correctly.

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

Keywords: WebAssembly, managed language, Dart, Haskell, OCaml, Ruby, Scheme

ACM Reference Format:
Manuel Serrano and Robert Bruce Findler. 2025. A Snapshot of the Performance of Wasm Backends for Managed Languages. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3759426.3760983

MPLR '25, Singapore, Singapore

ACM ISBN 979-8-4007-2149-6/25/10
https://doi.org/10.1145/3759426.3760983

Disclaimer

☞ This study contains no novelty regarding compilation or, more generally, implementation techniques. Its purpose is to provide information to language communities about the state of existing Wasm backends. Specifically, it offers information that sets some basic performance expectations and it offers some experience with the performance of several specific compilation technique choices in the context of the Bigloo Scheme compiler.

1 Introduction

WebAssembly [17, 47] (henceforth Wasm) is one of the newest virtual machines, aiming at providing a universal execution platform. Despite its name suggesting a design tuned for the web, it is a general-purpose assembly language, as found in other virtual machines such as the JVM [20]. It is strongly typed, memory safe, sandboxed, and does not support non-structured control flow operators (only loops and functions). It distinguishes itself from other virtual machines by being language-agnostic, closer to the hardware platform, and supporting only a minimal built-in runtime system.

The initial Wasm specification [28] provides a minimal set of features required for imperative languages with manual memory management, like C and Rust. As planned, after its initial release, Wasm has been extended with support for automatic memory management, polymorphism, function references, and exceptions [1]. These extensions make it suitable for managed programming languages. In this paper, we capture a snapshot of the performance of Wasm-based managed language implementations, focusing on:

- *Language coverage*: Is Wasm rich enough to be a possible target for many managed languages?
- *Execution speed*: How do current Wasm-based language implementations perform when compared to other execution platforms?

Overall, our findings are positive. Wasm is a robust platform and it is now a suitable time for language communities to engage in building implementations on top of it. That said, there is more work to be done as, generally speaking, the Wasm-based implementations lag behind the native implementations in both performance and functionality. Indeed, putting effort into a language implementation on Wasm now is likely to be impactful for the future directions of Wasm.

The rest of the paper is organized as follows. In Section 2 we introduce the Wasm platform. This section is intended for readers with little knowledge of Wasm. In Section 3 we evaluate the performance of various managed languages in comparison with native compilers for these languages. In Section 4 we present the new backend we have added to the Scheme Bigloo compiler [30]. We use that backend to evaluate various specific choices for generating Wasm code. We also present the limitations and difficulties we have faced when implementing that new backend. In Section 5 we discuss some constraints imposed by Wasm and their impact on the performance. In Section 6 we briefly discuss related work and in Section 7 we conclude.

2 WebAssembly 101

Wasm is a virtual machine *and* an intermediate language. This section offers a brief overview of the Wasm language. This section contains no novelty nor contribution of ours, but is included for the paper to be self-contained and to help readers who are unfamiliar with Wasm to understand the evaluation sections that follow.

2.1 Core Language

Wasm shares features of both high-level and assembly languages. Like assembly languages, Wasm has low-level arithmetic and a stack-based architecture. Like high-level languages, Wasm has global variables, local variables, functions and function calls, and nested expressions. Wasm does not expose registers as normally found in assembly languages and the stack is not a first-class object that can be manipulated explicitly by the program. Also, Wasm is type-safe.

Wasm has four primitive numeric types: 32-bit and 64-bit integers and floats. Booleans are implemented as 32-bit integers, with zero representing the false value. Wasm function parameters and results are typed.

Wasm supports several official syntaxes that are all compiled the same way. In this paper, we use exclusively the s-expression syntax. For instance, using this syntax, the Wasm definition of the classic `fib` recursive function is:

```
(func $fib (param $x i64) (result i64)
  (if (i64.lt_s (local.get $x) (i64.const 2))
     (then (return (i64.const 1)))
     (else
       (return
         (i64.add
           (call $fib
             (i64.sub (local.get $x) (i64.const 1)))
           (call $fib
             (i64.sub (local.get $x) (i64.const 2))))))))
```

The overall structure of `fib` is self explanatory but there are a few details worth pointing out. First, all arithmetic operations are strongly typed and there is neither type promotion nor any type casts. Second, expressions are nested which, in most situations, avoids explicitly pushing to or popping from the stack. Finally, the control flow operators are unusual for an assembly language as Wasm does not represent the program as a control flow graph made of basic blocks. It supports only `loop`, `if` forms, and jump tables.

Here, as a second example, is a program that reverses the elements of an array, demonstrating Wasm's control flow.

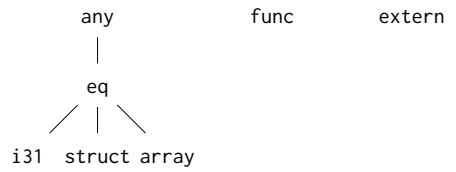
```
(func reverse
  (param $a (ref $i64arr)) (param $i i32) (param $j i32)
  ;; reverse the elements of the array $a from index $i to index $j
  (local $tmp i64)  ;; a temporary variable for array elements
  (loop             ;; an infinite loop
    (if (i32.lt (local.get $i) (local.set $j))
       (then        ;; swap the two array elements
         (local.set $tmp
           (array.get $i64arr $a (local.get $i)))
         (array.set $i64arr $a
           (local.get $i)
           (array.get $i64arr $a (local.get $j)))
         (array.set $i64arr $a
           (local.get $j) (local.get $tmp))
         (local.set $i
           (i32.add (local.get $i) (i32.const 1)))
         (local.set $j
           (i32.sub (local.get $j) (i32.const 1))))
       (else        ;; when the two indicies meet,
         (br 1))))) ;; exit from the loop
```

In this example, the `loop` form introduces an infinite loop that is eventually interrupted with `(br 1)` form, where the `1` means escaping the nearest enclosing loop, i.e. branching to the control block that's one layer outside the nearest enclosing block. In contrast, a `(br 0)` would branch to the start of the loop.

2.2 Wasm MVP

Wasm is built around the *Minimum Viable Product* principle. The initial specification described a minimal language large enough to compile C-like languages [28]. Over time, extensions have been added that make it a suitable platform for managed languages, the two most critical being the extensions for automatic memory management and exceptions.

Wasm represents the memory as a set of linear chunks of bytes that a program allocates *à la* C's `sbrk` [34]. These chunks are also used for communication with the outside world as we'll see in §2.3. The Wasm GC extension specifies a distinct memory allocator. It exposes a high-level view of memory allocation and memory references. The GC extension [3] enables programs to allocate and manipulate objects via *references*. The actual object memory layout is undisclosed, meaning that object access is fully controlled by Wasm itself. The GC extension provides the following initial type hierarchy:



The GC extension provides means for allocating structures and arrays and for accessing them. Like the memory layout, the actual implementation of these operations is opaque.

For the sake of an example, here are the definitions of a Lisp-like "cons" and "length" functions that respectively create a cons cell and compute the length of a list of cons cells. These definitions support well-formed lists only, that is, lists whose last element is the empty list, represented by Wasm's *null* value here.

```
(type $pair
  (struct (field $car (mut eqref))
          (field $cdr (mut (ref null $pair)))))

(func $cons
  (param $car eqref) (param $cdr (ref null $pair))
  (result (ref $pair))
  (return (struct.new $pair
            (local.get $car) (local.get $cdr))))

(func $length (param $l (ref null $pair)) (result i64)
  (if (ref.is_null (local.get $l))
     (then (return (i64.const 0)))
     (else (return (i64.add (i64.const 1)
                    (call $length
                      (struct.get $pair $cdr
                        (local.get $l))))))))
```

The `$pair` type declaration (user-declared Wasm identifiers are prefixed with '`$`') creates a subtype of `struct`.

The GC extension supports *null* values. They are so ubiquitous in Wasm, and the type `eqref` is an alias for "`(ref null eq)`", while the type "`(ref eq)`" denotes *non-nullable* values. Accessing a nullable value implies a dynamic test.

The `i31ref` type represents tagged 31-bit integers. These values can be converted to and from 32-bit or 64-bit integers. They are used to implement tagged non-boxed polymorphic integers, as in most implementations of managed languages. In statically typed languages, they can also be used as small scalar types, such as booleans or characters.

The `funcref` type enables pointers to functions. These are not closures as Wasm does not support local function definitions, but they are enough to let a compiler implement full-fledged closures.

```
(type $procedure (struct (field $entry (ref func))
                         (field $env (ref null array))))
(type $func1 (func (param (ref $procedure)) (param eqref)
                   (result eqref)))

(func $map (param $proc (ref $procedure))
           (param $l (ref null $pair))
           (result (ref null $pair))
  (if (ref.is_null (local.get $l))
      (then (return (ref.null none)))
      (else (return_call $cons ;; tail-call
              (call_ref $func1
                (local.get $proc)
                (struct.get $pair $car (local.get $l))
                (ref.cast (ref $func1)
                  (struct.get $procedure $entry
                    (local.get $proc))))
              (call $map
                (local.get $proc)
                (struct.get $pair $cdr (local.get $l)))))))
```

Some languages require unrestricted proper tail calls. To keep implementations from resorting to one of the numerous (and complex) techniques that are used when the underlying platform does not directly support tail calls, Wasm directly supports tail calls. Syntactically, tail calls appear as a variation of the `call` instruction, which is intended to be used in tail position with respect to the enclosing function.

2.3 The Outside World

Wasm is a small language which provides hardly any library functions but it supports two forms of external communication: first, the wasi [11] extension provides some posix-like functionality for server-side applications, and second, the JavaScript [13] foreign function interface, which is the natural option for executing code from within web browsers.

One of the obvious uses of the JavaScript interface is for taking advantage of natively supported features such as math and string operations. For instance, Wasm supports 64-bit floating point numbers but it provides only basic arithmetic. Math functions such as `cos` and `sin` are not supported. Fortunately using the JavaScript implementation from within Wasm is straightforward, but it does involve crossing the boundary between the two languages twice.

In some other situations the exchange involves not only integers or floating-point numbers. For that, the GC extension provides the `externref` type. It enables opaque values to transit in Wasm garbage collected code. It is generally used to let a Wasm GC-allocated object point to a JavaScript object. A common example of using `externref` is for bignums.

3 Multilingual Experiment

The experiment presented in this section attempts to shed light on how suitable and performant Wasm is for executing a variety of managed languages.

Managed languages come in all sorts of flavors: side-effecting vs pure, lazy vs eager, support for full continuations vs only enough for effect handlers, a fully concurrent runtime system vs an assumption of single-threading, etc. All of this variety imposes different constraints on the execution platform and raises difficult-to-answer questions about which languages should be included in our experiments. In an effort

	typing	evaluation	concurrency
Dart	static	strict	isolates
Haskell	static	lazy	green threads
Bigloo	static/dynamic	strict	pthreads
Hoot	dynamic	strict	green threads
OCaml	static	strict	domains
Ruby	dynamic	strict	global lock

Figure 1. Language main features.

to make impartial decisions, we have limited our experiment to the language implementations that satisfy the following criteria:

1. Being a real language: This notion is obviously subjective so we took the simple definition that the language must be listed in the 100 most popular programming languages according to the Tiobe index [38].
2. Having a robust implementation: Concretely, we define this to mean that the Wasm compiler is based on a widely used implementation for the language.
3. Having a mostly pain-free automatic installation process: This criterion is also loose but we limit our experiment to languages for which the effort to create installation scripts is reasonably simple and does not require extensive script tweaking. For context, the scripts that install the various languages are at most 20 commands.

We have found 6 implementations satisfying these criteria: Dart [12], Haskell [32] (ghc [15]), OCaml [45] (wasm_of_ocaml [42]), Ruby [48], and Scheme [19] (guile [9] and bigloo [30]); their characteristics are summarized in Figure 1.

For each of these languages, we compared the performance of the Wasm implementations with the corresponding widely-used implementation. For all of these implementations, the new Wasm code generator is implemented as a new backend added to an existing compiler and we take advantage of that to make the experiments more meaningful. Because the compilers share the front end, the comparisons are more generally informative about Wasm, since the source language and the middle end of the compiler is the same (although not all features are always supported).

With the exception of OCaml, each of the Wasm implementations is based on the best-performing implementation of the language. OCaml, however, is based on the OCaml byte-code compiler, not the native compiler; section 3.5 explains the situation in more detail.

In addition to many Wasm-based languages, there are also many Wasm engines available. They evolve quickly but, as of the date of this report, only a few of them support the whole set of features needed for the broad set of managed languages we consider. Specifically, the "Garbage Collection" extension (and the related "Reference Types" and "Typed Function References" extensions), the "Tail Call" extension, and the "Exception Handling with exnref" extensions are essential. This limits the set of engines to Google's V8 [16], Mozilla's SpiderMonkey [24], and Apple's JavaScriptCore [8]. Our experiments reveal that V8 is the most robust and feature-complete, so we use it for most of the experiments. SpiderMonkey is also almost feature-complete but its host environment is not as rich as Node.js's. Consequently, much of the code generated by the compilers is incompatible with it. Still, we ran the Dart and Bigloo benchmarks with SpiderMonkey, as shown in the respective sections. Unfortunately, even though JavaScriptCore officially supports all of the extensions, the essential GC extension required for this experiment is available only in the Safari port and does not yet ship in WebKitGTK, which we need for running the benchmarks on Linux.

In the following subsections, we give an overview of all these language implementations and compare the performance of their compilers on standard benchmark suites for the languages.

The experiments presented in this paper have all been conducted on Linux according to the same protocol:

- Benchmarks are executed 10 times and wall clock execution times are collected. Repetitions of each variant and baseline are paired in order of execution to compute relative execution time for each pair. The figures report geometric means of these relative times, with geometric standard deviations.
- A single platform was used: an AMD Ryzen Threadripper PRO 7955WX 16-Cores running Debian Linux 6.12.33+deb13-amd64 x86_64 with 61 GB of memory.
- Wasm programs are executed within Node (v24.4.1, powered by V8 v13.6.233.10-node.17) with a minimal stack size of 8MB and, when compatibility permits, they are also executed by SpiderMonkey C128.13.0 using the `-wasm-compiler=optimizing` option. Executions without any mention of the engine refer to V8.

3.1 Dart

Dart's Wasm implementation ships with its standard distribution, so it is easy to install, requiring no external tools. A single compiler can generate native code, JavaScript code, and Wasm code, making it an ideal candidate for comparing the performance of the three execution platforms. The generated Wasm code requires the GC extension and it targets the JavaScript environment.

To evaluate Dart's performance, we selected benchmarks from the shootout suite [10], which mostly consists of small programs and microbenchmarks. The performance results delivered by the three backends are presented in Figure 2. The JavaScript backend fails on coro-prime-sieve.

The performance of the backends varies significantly. One benchmark, fib executes almost 2x faster than the native backend. The two benchmarks that use big integers edigits and pdigits are significantly slower than the native backend. As the JavaScript backend also executes slowly these two

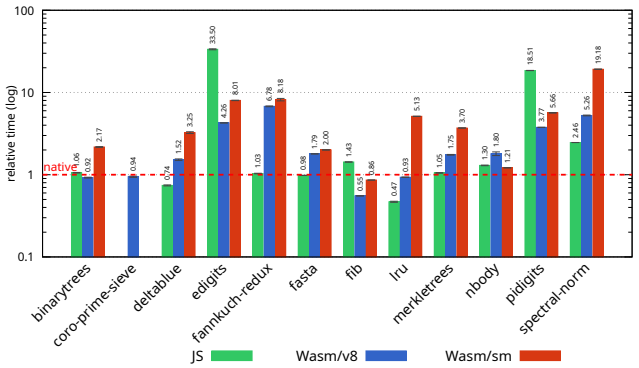


Figure 2. Execution time comparison between Dart (v3.8.2) native, Dart JavaScript, and Dart Wasm. Wasm programs are executed by V8 and by SpiderMonkey. The dotted red line indicates the performance of native code; lower means faster execution than the native version. A value of 0 means the test failed. Logarithmic scale used.

tests, this suggests that maybe the JavaScript and Wasm backends do not use the native JavaScript bigint primitive type. The Wasm backends are no more than 2× slower than the native backend on the other benchmarks. Finally, we observe that, with the exception of the binarytrees, lru, and spectral-norm, SpiderMonkey is at worst around 2x slower than V8 and on one test, nbody, SpiderMonkey is faster than V8.

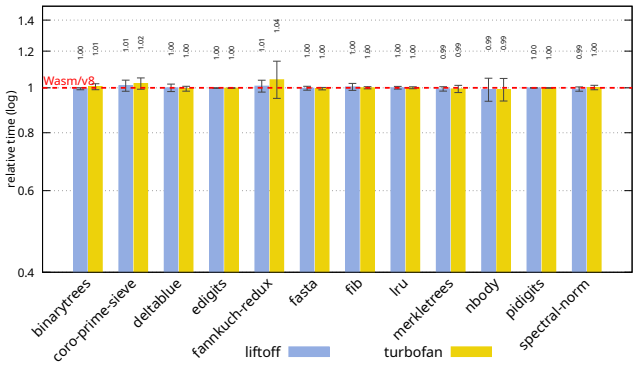


Figure 3. Execution time comparison of Node execution pipelines when executing Dart (v3.8.2) Wasm benchmarks. The dotted red line indicates the default pipeline. Logarithmic scale used.

Node supports several execution pipelines [2]. We compare their performance in Figure 3. As we did not observe a significant impact when using one pipeline or another, for the rest of the experiments presented in this paper we use the default execution pipeline.

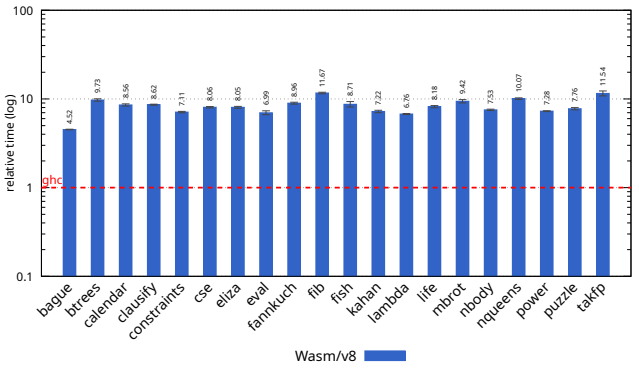


Figure 4. Execution time comparison between ghc (v9.6.6) native and ghc Wasm. The dotted red line indicates the performance of native code; lower means faster execution than the native version.

3.2 Haskell, GHC

The GHC Wasm backend is still a tech preview and not included in the official repository yet. As such, it currently lacks an automatic installation procedure. The documentation about the GHC Wasm backend is scarce and most of the available online documentation predates its integration into the main GHC development tree [31]. The GHC Wasm was developed before the integration of the GC and `funcref` Wasm extensions. GHC's Wasm implementation uses the same garbage collector as the standard implementation. It is implemented in C and compiled to Wasm.

Figure 4 compares the performance of the GHC Wasm backend to that of the native backend. The performance ranges from 4× up to a little more than 11× slower than the native performance. The penalty for not using Wasm's newer features is significant but it should be noted that the backend is robust, as all benchmarks execute without error.

3.3 Scheme, Bigloo

Bigloo[1] is a variant of Scheme [19]. It lacks the full support of tail recursion that Scheme requires, but extends Scheme with object-oriented programming based on single inheritance classes and generic functions, exceptions, multi-threading, deep embedding of a host language, and optional type annotations. Types can denote Scheme values, *e.g.*, `pair` or `vector`, as well as native values, which depend on the compiler backend. For instance, when a module is compiled to C, types may denote C values such as "`double`" or "`char *`". The compiler ensures type safety by inserting guards and conversions in the generated code; the compiler also supports separate compilation.

The latest unstable Bigloo version (5.0a) ships with three backends: C, Jvm, and Wasm. The latter is still under development. It lacks features that require newer Wasm extensions,

[1] https://www-sop.inria.fr/indes/fp/Bigloo

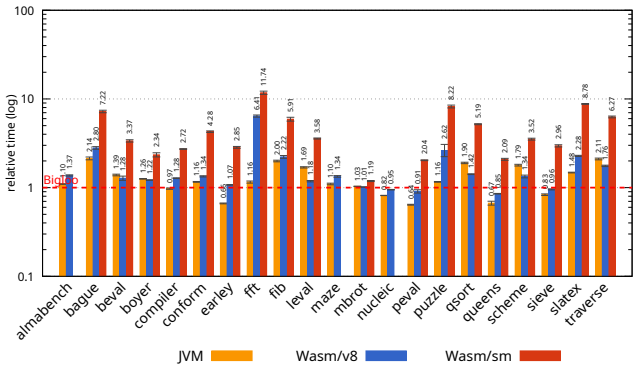


Figure 5. Execution time comparison between Bigloo (v5.0a) native, Bigloo Jvm, and Bigloo Wasm. Wasm programs are executed by V8 and by SpiderMonkey. The dotted red line indicates the performance of native code; lower means faster execution than the native version. Logarithmic scale used.

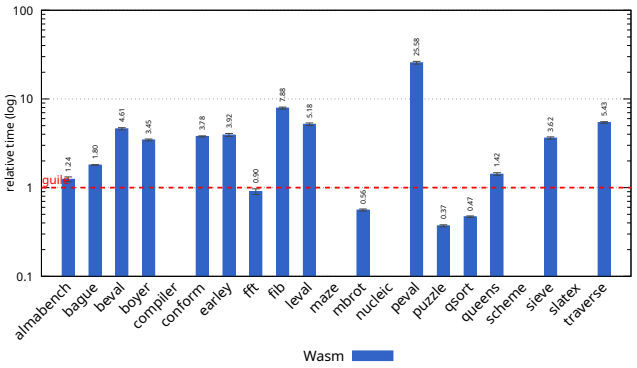


Figure 6. Execution time comparison between Guile (v3.0.10) native and Hoot (v0.6.0). The dotted red line indicates the performance of native code; lower means faster execution than the native version. A value of 0 indicates a test failure. Logarithmic scale used.

such as full `call/cc` support, multi-threading, and system and network programming.

Bigloo compiles Scheme functions as Wasm functions and function calls use the Wasm stack. It uses the GC, exceptions, tail-call annotations [23], and the `i31ref` Wasm extensions. The compiler generates either `.wat` files or `.wasm` binary files.

Figure 5 compares the performance of the three backends. This experiment used the bglstone benchmark suite [22]. First, we observe that all the benchmarks execute with the experimental backend, meaning that Wasm is rich enough to implement all of the features of a Scheme-like language except full-fledged `call/cc` (which none of the benchmarks use). Second, we observe that the performance of the Wasm backend under V8 is usually slower than the C backend but is within 10% (sometimes faster and sometimes slower) on 6 tests. SpiderMonkey is consistently slower than V8 but by less than 5x on all of the tests. We also observe that apart from a few benchmarks the Wasm backend is also slower than the JVM backend but with a smaller ratio.

3.4 Scheme, Hoot

Hoot [44] is a new Wasm backend for Guile [9]. According to its website,[2] the version 0.6.0 that we use for this report supports most of the language without any particular restriction. In particular continuations are not mentioned to suffer any restriction. Hoot requires the Wasm GC and tail-call annotations extensions. The compiler uses explicit continuation passing style (CPS) conversion [7, 26] and stack allocates return continuations. The Hoot compiler compiles the whole program at once and produces a single Wasm binary file. We use the same set of Scheme benchmarks as we did for Bigloo (Section 3.3). The results are presented in Figure 6.

[2] https://www.spritely.institute/news/guile-hoot-0-6-0-released.html

The first observation is that 5 out of the 21 benchmarks fail with various runtime errors. The three benchmarks that are float intensive (almabench, fft, and mbrot) are fast with Wasm. This suggests that the Wasm GC is a significant part of the execution as floating-point numbers are generally implemented using boxed numbers. Also, the puzzle and qsort benchmarks are faster than the native backend. For the other benchmarks, Wasm's slowdown has a wide range, with ratios ranging from about 1.2× to about 25×. The Guile Wasm backend imposes some of the biggest penalties among all the implementations we tested. Lacking a technical description of the compiler, we can only conjecture about the reasons for such a gap. We suspect that the full support of continuations (that Bigloo, the other Scheme-based implementation we tested, does not support) uses some compilation techniques and encodings that are difficult to implement efficiently in Wasm. It may also be that most of the optimizations that Hoot inherits from the Guile compiler are redundant with those of the Wasm engine and as such have little to no effect.

3.5 OCaml, wasm_of_ocaml

The official OCaml distribution contains two compilers, an optimizing native compiler, ocamlopt, and a byte-code compiler, ocamlc. The two have a high degree of compatibility but use entirely different implementations; the native compiler produces significantly more efficient code at the cost of a longer compilation time. The Wasm implementation of OCaml, Wasm_of_ocaml, is a third-party compiler that is based on the byte-code compiler. To keep the focus on the performance of the Wasm backends, we treat the byte-code compiler as the baseline, as it shares the most code with the Wasm implementation. For context, however, we also report the performance of the optimizing native compiler. Further complicating matters, the Wasm_of_ocaml compiler comes

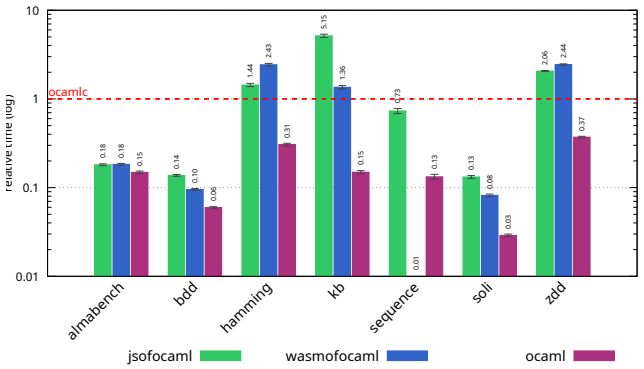


Figure 7. Execution time comparison between OCaml (v5.3.0) native, OCamlc (byte code interpreter), Js_of_ocaml (v6.1.1) (byte code compiled to JavaScript), and Wasm_of_ocaml (v6.1.1). The dotted red line indicates the performance of OCaml code; lower means faster execution than the native version.

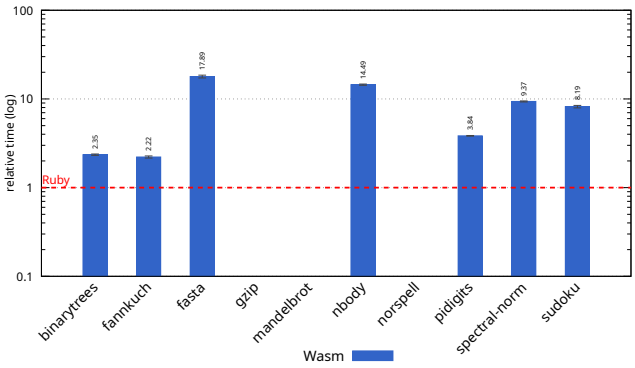


Figure 8. Execution time comparison between Ruby (v3.3.8) native and Ruby Wasm. The dotted red line indicates the performance of native code; lower means faster execution than the native version. A value of 0 indicates a test failure. Logarithmic scale used.

with (and is based on) Js_of_ocaml, a compiler that generates JavaScript code directly from the OCaml byte-code. Because it seems interesting, we also report on the performance of Js_of_ocaml. Figure 7 presents the relative performance of all these compilers.

In terms of performance, wasm_of_ocaml seems to be an interesting alternative to js_of_ocaml, as it performs at least as well for all but one test (hamming) and on sequence it even dramatically outperforms the optimizing native OCaml compiler. It is also an interesting alternative to ocamlc, being as fast or faster on the majority of tests with the worst overhead being less than 2.5×. The overhead with respect to the native compiler is probably still too significant to be acceptable, except for applications that derive significant benefit from running in the browser.

3.6 Ruby

Ruby.wasm is a Wasm port of CRuby. For this experiment we used version 3.3.8, the most up-to-date version at the time of writing. Apart from the lack of thread and networking support, no other limitations are reported.

Figure 8 shows the performance comparison between CRuby and ruby.wasm. The port is not fully operational, with one test crashing (mandelbrot) and two tests not running because of wasi incompatibilities (gzip and norspell). The performance impact is significant, but since technical details are not publicly available, we cannot provide an informed explanation.

3.7 Conclusion

In this section we report measurements of the performance of the Wasm backend for 5 languages and 6 different systems. Although there are programs where the Wasm performance exceeds the native performance, we observed that, generally speaking, Wasm performance is not yet comparable to that of native implementations. It is not uncommon to observe a penalty ranging from 4× to 10× or even more. We also observe that only half of the systems we tested can execute all the benchmarks successfully. Considering that V8, the Wasm platform we used for the experiment, is the most advanced one, this experiment suggests that Wasm platforms are not yet offering a performant alternative to native execution. Figure 9 offers a summary.

	succ.-fail	best	worst	geo. mean
Dart	12-0	0.55	6.79	1.90
Haskell	20-0	4.51	11.66	8.18
Bigloo	21-0	0.86	6.31	1.43
Hoot	16-5	0.37	25.57	2.42
OCamlc	7-0	0.01	2.44	0.27
OCamlopt	7-0	0.07	9.08	2.12
Ruby	7-3	2.22	17.89	6.31

Figure 9. Summary of Wasm ports. The **succ.-fail** column reports the number of benchmarks that respectively succeed or fail. The **best** and **worst** columns report the ratio of the fastest and slowest Wasm execution. The **geo. mean** column reports the geometric mean of the Wasm and native compiler.

4 Wasm Compilation Strategies

In this section we take advantage of the flexibility of the Bigloo compiler to explore different compilation paths and options and we evaluate some features of the Wasm platform.

4.1 Control Flow

The lack of a goto instruction, which prevents the program from being represented as a classical control flow graph with basic blocks, is one particularity of Wasm. It requires front-end compilers that use classical control flow graphs formed by connected basic blocks to transform the programs into nested loops and switches for Wasm. Any program can be transformed using a generic schema which consists of simulating gotos using a local variable and a switch [14]. In Wasm, this produces functions like this one:

```
(func $F (param $x (ref eq)) (result (ref eq))
   ;; use a local variable to hold a "__PC"; start in basic block #0
   (local.set $__PC (i32.const 0))
   (loop $__dispatcher
     (block $bb_2                                  ;; basic block #2
        (block $bb_1                               ;; basic block #1
           (block $$bb_0                           ;; basic block #0
              (br_table $bb_0 $bb_1 $bb_2
                 (local.get $__PC))
              ...
              ;; go to basic block #1
              (local.set $__PC (i32.const 1))
              (br $__dispatcher))
           ...
           ;; go to basic block #0
           (local.set $__PC (i32.const 0))
           (br $__dispatcher))
        ...)))
```

Algorithms, generally referred to as *reloopers*, that avoid using this dispatcher mechanism have been extensively studied [6, 25] but their implementation is complex and they are not always practical because they can significantly increase the size of the generated code. This raises the question whether relooper algorithms should be used. Figure 10 compares the performance of the dispatch-based compilation scheme and the relooper algorithm. It clearly establishes the huge benefit of relooper which should undoubtedly be used except possibly for pathological programs that grow excessively, as the relooper algorithms can be exponential in the worst case.

4.2 Tail Calls

Tail-calls were added to Node.js in 2023, with Node version 11.2 [23]. This improvement, although essential for functional languages, might negatively affect the overall performance because it requires complex stack frame manipulation. Figure 11 compares the performance of Wasm when tail calls annotations are used or not used. As we can see, the slowdown is marginal as all but one benchmark are not impacted. This suggests that the tail-call annotation is ready for a large adoption and that it should be used by all systems.

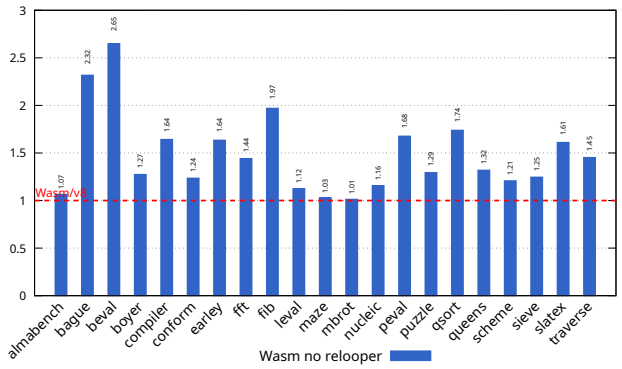


Figure 10. Execution time comparison between relooper control flow compilation and generic compilation. `1.0` indicates the performance of *relooper* code; lower is faster. Linear scale used.

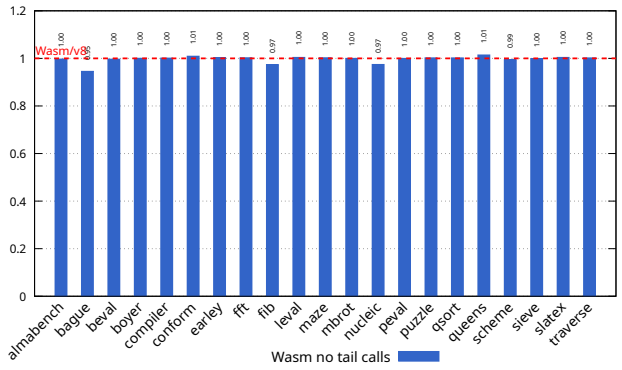


Figure 11. Impact of tail-call annotation on execution speed. `1.0` indicates the performance of Wasm code *with* tail call; lower means faster execution. Linear scale used.

4.3 Front-End Register Allocation

Wasm is a stack machine with temporary variables being mapped to stack frame slots. A front-end compiler may declare as many temporary variables as it needs for holding local expressions or it may also implement an optimization like register allocation to map these temporary variables to a smaller subset of Wasm temporaries. The Bigloo compiler can generate code both ways. It can run a register allocation that minimizes the number of temporary variables by mapping several variables of the same type to a single, well-typed temporary. We use this flexibility to check if minimizing the number of temporaries is currently beneficial or if the official Wasm tool chain manages to use physical registers efficiently.

Figure 12 measures the impact of the front-end register allocation. This experiment shows that there is a minor benefit in mapping temporaries to a reduced set of variables as

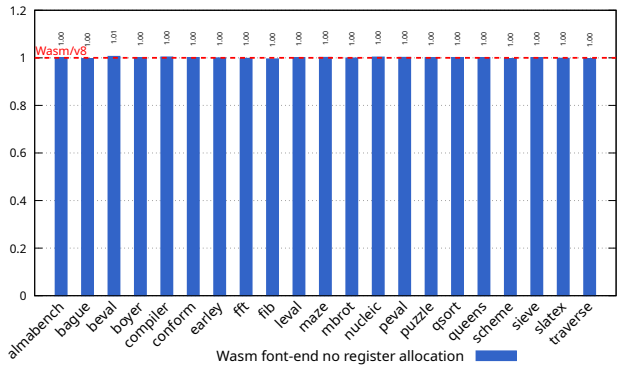


Figure 12. Impact of front-end register allocation. `1.0` indicates the performance of Wasm code; lower means faster execution than the native version. Linear scale used.

two tests show minor improvements (bague and beval) and one shows a significant improvement (puzzle).

4.4 Fixnum Arithmetic

A compiler targeting Wasm has two options to implement small integers. It can either box them or use the `i31ref` type. The former has the benefit of supporting full range 32-bit or 64-bit integers but the drawback of requiring memory allocation. The latter has the benefit of avoiding memory allocation but it restricts the range of integers to $[-2^{31}..2^{31}-1]$. The Bigloo Wasm backend can be configured either way.

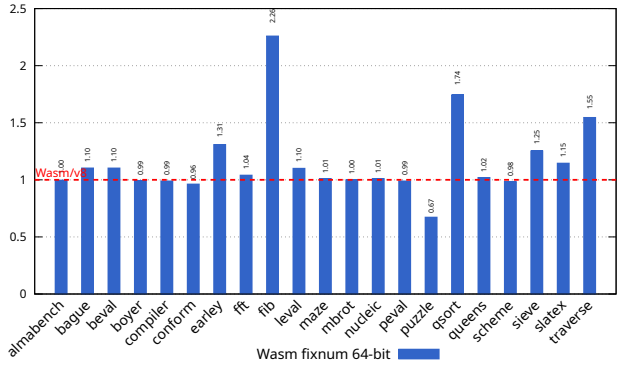


Figure 13. Impact of fixnum arithmetic. `1.0` indicates the performance Wasm `i31ref` code. Lower is faster. Linear scale used.

Figure 13 compares the performance of boxed integers. The benefit of using `i31ref` is significant for a minority of benchmarks and there are no significant penalties. This is because for most tests, there are few distinct integers used and these values are pre-allocated at initialization time. The performance benefit of `i31ref` is not so important that it eliminates the option of using boxed 64-bit integers. The option to choose probably depends on each system. For Bigloo, the data suggested to use `i31ref` as default configuration (but boxed integers can be selected at installation time).

5 Wasm Compilation Idiosyncrasies

The recent garbage collection and the exception mechanism extensions make Wasm an interesting target for managed languages. However, some bits are still missing for efficient and complete support of all features that managed languages use. In this section, we detail the main obstacles.

5.1 Safety

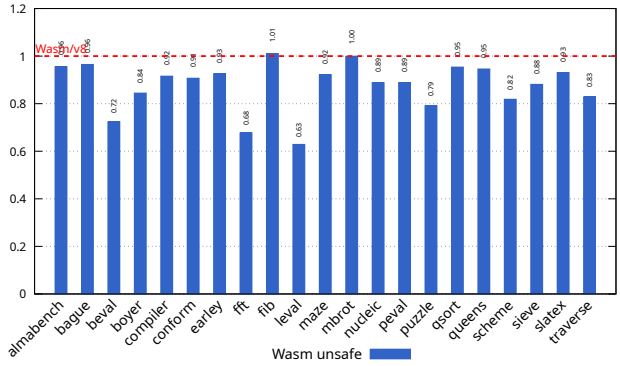


Figure 14. Impact of runtime guards on Node.js Wasm implementation. `1.0` indicates the performance of safe guarded code; lower means faster execution than the native version. Linear scale used.

Wasm is a *safe* platform with strict type enforcement. At compile time, the Wasm assembler checks the consistency of variables types, type casts, and correct execution stack balance. At run time, if a dynamic cast fails or if an index overflows an array bound, an exception is raised. These safety guarantees are implemented via run time tests, which obviously reduce the execution speed. This reduction is evaluated in this section.

Node.js supports *unsafe* executions where all dynamic guards are disabled and all dynamic casts are assumed to succeed. We used this feature to evaluate the cost of the Wasm dynamic guards. Figure 14 compares the performance of Bigloo executions when the Wasm dynamic safety tests are enabled or disabled.

Overall, the impact of dynamic tests is generally significant and, for three benchmarks, disabling dynamic tests even enables the Wasm backend to outperform the native backend.

5.2 Runtime Types

The Wasm GC extension is built on top of a type hierarchy (see §2.2) that supports subtyping and dynamic type checking. This constitutes an appropriate and efficient mechanism for verifying run-time types. As much as possible, this is

what Bigloo uses for built-in types. For instance, Scheme pairs are mapped to a Wasm `struct` with a `car` and `cdr` fields and the predicate `(pair? e)` is implemented as `(ref.test (ref $pair) e)`. Unfortunately this efficient type encoding is not always possible.

Wasm uses *mutually iso-recursive types* [27], meaning that two types are considered equivalent if their positions in equivalent blocks are the same. For instance, in

```
(rec
   (type $ta (struct (field $a i32))))
(rec
   (type $tb (struct (field $b i32))))
```

the types `$ta` and `$tb` are equivalent, but in

```
(rec
   (type $dummya (struct))
   (type $ta (struct (field $a i32))))
(rec
   (type $dummyb (struct (field $__dummy i8)))
   (type $tb (struct (field $b i32))))
```

they are different, because the types `$dummya` and `$dummyb` make the two recursive blocks different. Using the native type checks requires that the Scheme types are all mapped to different iso-recursive Wasm types. While a careful implementation of the built-in types (pair, vector, string, fixnum, flonum, etc.) can enforce the uniqueness of the blocks by encapsulating them in different Wasm `rec` constructs and by carefully using dummy unique types in each block, this is not possible for compiler-generated types. This is particularly critical when implementing a class-based object-oriented language, as Bigloo is.

The compilation of a Bigloo class produces a host structure with a private header plus fields corresponding to the class properties. For instance, the following classes:

```
(module example
   (export (class point x::double y::double)
           (class point3d::point z::double)))
```

compile to:

```
(module $example
   (rec
      (type $point
         (sub $object
            (struct
               (field $header (mut i64))
               (field $x (mut f64))
               (field $y (mut f64)))))
      (type $point3d
         (sub $point
            (struct
               (field $header (mut i64))
               (field $x (mut f64))
               (field $y (mut f64))
               (field $z (mut f64)))))))
```

Unfortunately, unlike the built-in types, the class type predicates cannot be mapped to simple Wasm type tests. In addition to the `point` class above, let us imagine another Bigloo class `complex` also declaring two floating point fields. For `point` and `complex` to correspond to two different Wasm types, they have to be declared in different `rec` contexts. In other words, the Bigloo compiler has to forge two different contexts for declaring `point` and `complex` but when these two classes are declared in different modules compiled separately, this is difficult. It requires a sophisticated link pass that, currently, Bigloo does not implement. As a consequence, testing that a value is an instance of a particular class requires two checks. First, checking that it is an object using the Wasm built-in test and, second, that it is also an instance of the desired class.

5.3 Variable Initialization and Nullable Values

Wasm type safety requires that all local variables are either declared as *nullable* and checked at each use or pre-initialized before being used. Using nullable types loses most of the advantages of a strongly typed target language and pre-initializing local variables is difficult to avoid because Wasm does not consider that the first block dominates the rest in a sequence of blocks. For instance, the following program is rejected because, at the `return` statement, the variable `$a` is considered to be uninitialized.

```
(func $main (export "__main")
   (result i32)
   (local $a (ref $ta))
   (if (i32.const 1)
       (then (local.set $a (struct.new $ta (i32.const 1))))
       (else (local.set $a (struct.new $ta (i32.const 2)))))
   (return (struct.get $ta $a (local.get $a))))
```

A simple solution for correcting this program is either to duplicate and lift the `return` statement in the two branches of the conditional or to pre-initialize the variable `$a` with a default value of type `$ta`. This is a major concern for the front-end compiler because, even if it can prove using simple def-use analysis that all references to a local variable are safe, it cannot always generate code that avoids the unnecessary pre-initialization. This problem has been discussed at length and is well known by the Wasm developer community [4, 5].

For Bigloo, we tried both options. Before generating the final Wasm code, the compiler analyzes each function using the Wasm dominator analysis and, if a variable is considered to have possibly been used before initialization, it either introduces a pre-initialization with a default value or uses a nullable type.

We conducted an experiment to compare the performance of the two options. Its result is presented in Figure 15. It shows marginally better results when using default pre-initialization to avoid nullable types.

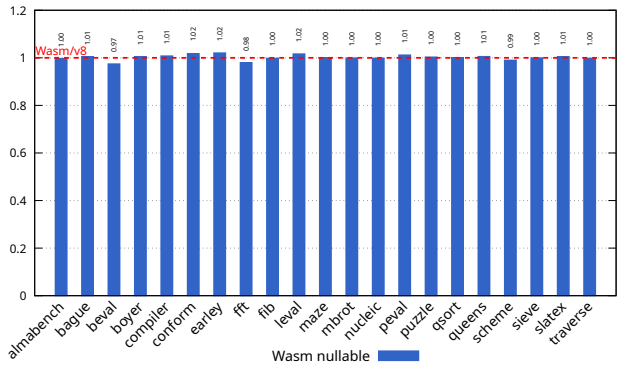


Figure 15. Impact of nullable types on execution speed. `1.0` indicates the performance of pre-initialization to avoid nullable types. Lower means faster execution. Linear scale used.

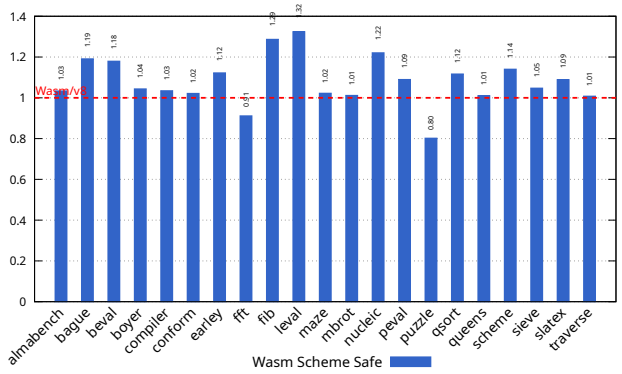


Figure 16. Impact of double type checks on execution speed. `1.0` indicates the performance of Wasm code without explicit Scheme type checks. Lower means faster. Linear scale used.

5.4 Redundant Type Safety and Exceptions

Scheme is a safe language, as most managed languages are. Bigloo uses a typed intermediate representation of the program and as much as possible it emits typed functions and typed variables. When it cannot prove that an access is type safe, it generates guards. For instance, without any external context, implementing a pair access `(car v)` would be compiled into `(if (pair? v) (unsafe-car v) (error))` which, in Wasm would be written:

```
(if (ref eq) (ref.test (ref $pair) (local.get $v))
    (then (struct.get $pair car (local.get $v)))
    (else (throw $fail)))
```

Unfortunately this code is rejected by Wasm because the reference of the variable `$v` has to be explicitly cast to a pair in the `car` access. To solve that problem, the compiler actually generates the following code:

```
(if (ref eq) (ref.test (ref $pair) (local.get $v))
    (then (struct.get $pair car
            (ref.cast (ref $pair) (local.get $v))))
    (else (throw $fail)))
```

This involves a double test of the variable. Of course, one may use an *unsafe* execution flag of the Wasm VM when supported (see §5.1), or one may hope that an optimizing VM can eliminate the redundant test. This cost of these extra tests is measured in Figure 16. It compares the performance of Bigloo delegating all type tests to Wasm and the performance of Bigloo inserting type tests in addition to Wasm guards. This experiment shows that the current V8 Wasm implementation does not eliminate the double type checking, which has a significant impact on the performance.

Another approach would simply be to drop the surrounding conditional expression. As Wasm is safe, if the variable `$v` is not a pair, it will trigger an internal error. This might be an interesting option in the future but as of the beginning of 2025, the internal exception mechanism makes this impractical. In case of a type error, the system triggers a trap that goes to JavaScript instead of a catchable exception. As such, it is impossible for a client program to intercept the error and handle it inside the Wasm program.

The Wasm `br_on_cast` instruction can be used to remove redundant checks but the occurrence typing technique [39] (that compilers of managed languages frequently use) requires that the type check itself be separate from the guarded built-in operation, making the reconstruction of an expression using `br_on_cast` difficult. For that reason, as of the writing of this paper, the Bigloo compiler cannot take benefit of that Wasm instruction and requires the double type checks mentioned above.

5.5 Missing Features

Other than performance issues, the Wasm specification still lacks some features required to implement all features commonly found in managed languages. In particular, for the Bigloo system it still missed four features:

1. `call/cc`: until the stack manipulation specification is completed and fully implemented, full continuations cannot be implemented efficiently without hurting the performance of the rest of the language. Currently, Bigloo supports continuations that are used only in their dynamic extent to escape from a computation, much like exceptions commonly found in other languages. In contrast, Hoot supports full continuations. This difference might be the reason why Hoot imposes a larger penalty over the native backend than Bigloo does.
2. No modules linking. Wasm does not support native module linking. It supports communication with an external host, JavaScript in our case, but not a mechanism that would let two separately compiled Wasm

modules communicate either via function calls or efficient variable sharing. Currently this can be simulated via external JavaScript function calls but this feature is not part of the official Wasm standard yet.
3. In a similar vein, Wasm does not support shared libraries nor dynamic loading of object files. This also implies long compilation times, as all object files have to be linked together before execution. This limits the performance of `eval` functions that cannot generate and load efficient Wasm code.
4. Wasm is an isolated platform. It can communicate with a JavaScript host or use posix-like extensions, but it cannot be linked against an arbitrary native library. For instance, it would be difficult for a Wasm compiled application to use an existing SSL library or multimedia libraries. This is a major burden for languages that are shipped with an extensive set of libraries, as Bigloo is. For Bigloo, this phenomenon accounts for the most significant burden of the Wasm port. The whole implementation of the new code generator is a mere 4,000 lines of Scheme code, but the still incomplete implementation of libraries is already more than 13,000 lines of hand-written Wasm code.
5. The JavaScript asynchronous nature makes it difficult to implement synchronous runtime system. As long as the Wasm JavaScript Promise Integration extension is not available it is difficult to implement blocking I/O operations.

6 Related Work

Zhang et al. [49]'s article *Research on WebAssembly Runtimes: A Survey* gives an overview of the main Wasm implementations, how they are described in academic publications, and what the main application domains for the language are. It gives a broad overview of ongoing Wasm-related research.

Most performance studies of Wasm concentrate on either a comparison with JavaScript performance [36, 40, 41, 46] or with C-like languages [18]. Some focus on the performance comparison inside browsers [35], while some focus exclusively on the server-side [33]. Some are even more specific in comparing the performance of JavaScript and Wasm on particular application domains [29, 43]. In contrast, this paper offers a broader view including 6 different managed languages and is agnostic with respect to server-side or client-side programming, although for the simplicity's sake, all experiments are conducted on the server side.

Szewczyk et al. [37] study the impact of the dynamic memory bound checks Wasm executes. They show that, regardless of the underlying hardware, the memory safety of Wasm incurs slowdowns that can be reduced to about 20% for C programs.

Mäkitalo et al. [21] propose a mechanism for adding dynamic loading to Wasm that slows down executions only marginally, which is interesting in the context of web pages and on-demand page load.

7 Conclusion

The latest evolution of Wasm makes it suitable for compiling managed languages. The platform still misses explicit stack manipulation, which makes support for continuations, that some languages need, complex and slow. Its lack of an efficient linking mechanism is also a current limitation of the system as high-level languages generally provide a rich runtime system with an extensive set of libraries that, in the context of Wasm, have to be packed alongside client programs. Multi-threading is also missing, which prevents some applications from being ported to Wasm. And finally, the expected "JavaScript promise integration" extension will make it possible to implement blocking and synchronous IO operations some systems demand. These are the main ingredients that Wasm still lacks for being a fully general platform. Future extensions, following Wasm's incremental approach to language extension may well fill these needs.

This experience report focuses on the performance of actual implementations of managed languages that have added or are adding a Wasm backend. It compares the speed of Wasm to that of pre-existing backends.

First, we observed that none of the Wasm implementations we tried matches the performance of native code. In the most favorable case, Wasm imposes a penalty in the range of 1.5× but it can go up to about one order of magnitude. We explored two languages that also support JavaScript backends, Dart and OCaml. The benefit of using Wasm over JavaScript is only visible for the latter. The Bigloo language can be compiled to native code, JVM bytecode, and Wasm code. The comparison of the three code generators shows that, on average, the Wasm backend is significantly slower than the native and Jvm backends. Although we cannot say for sure that this is a shortcoming of Wasm, as opposed to the Bigloo backend, Bigloo is a sophisticated and mature compiler, suggesting that the largest opportunities for improvement are inside Wasm itself.

Second, using the flexibility of the new Bigloo Wasm backend we explored possible variations of the code generator. We observed that the most crucial transformation a compiler must use is a relooper optimization that transforms a classical control flow graph into nested loops. The second most important option is to avoid, as much as possible, nullable types and instead pre-initialize variables with default values. Finally, we also observed that using `i31ref` for languages that can afford to limit the range of integer values improves the performance of arithmetic operations without being a true game changer. Statically typed languages that use `i31ref` for small scalar types may find a bigger benefit.

Acknowledgements

We offer a special thanks to Andreas Rossberg for his comments on a draft of this paper as well as his helpful insights about Wasm. Thanks also to Zubin Duggal and the reviewers for their comment on a draft of the paper.

References

[1] 2022. WebAssembly Core Specification. https://www.w3.org/TR/wasm-core-2/

[2] 2025. WebAssembly compilation pipeline. Retrieved 2025-07-31 from https://v8.dev/docs/wasm-compilation-pipeline

[3] Andreas Rossberg and the WebAssembly Community Group. 2023. GC Proposal for WebAssembly. GitHub repository "WebAssembly/gc". Retrieved 2025-07-30 from https://github.com/WebAssembly/gc Archived April 25, 2025; draft outlines struct and array heap=types.

[4] Anonymous. 2018. WebAssembly Troubles part 1: WebAssembly Is Not a Stack Machine. http://troubles.md/wasm-is-not-a-stack-machine

[5] Anonymous. 2021. Elaboration of let alternative option (6): Null checks on local.get. https://github.com/WebAssembly/gc/issues/187

[6] anonymous. 2021. WebAssembly Troubles part 2: Why Do We Need the Relooper Algorithm, Again? http://troubles.md/why-do-we-need-the-relooper-algorithm-again/

[7] Andrew W. Appel. 1992. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI)*. ACM, San Francisco, CA, USA, essence. doi:10.1145/143095.143140

[8] Apple Inc. 2002. JavaScriptCore. https://webkit.org/projects/javascript/. Accessed: 2025-07-30.

[9] Aubrey Jaffer and Tom Lord and Miles Bader and the GNU Project. 2024. GNU Guile: Ubiquitous Intelligent Language for Extensions. https://gnu.org/software/guile Version 3.0.10, released June 24, 2024; accessed 2025-07-30.

[10] benchmarksgame team. 2025. The Computer Language 25.03 fBenchmarks Game. Retrieved 2025-07-30 from https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html

[11] Bytecode Alliance. 2019. WebAssembly System Interface (WASI). https://wasi.dev. Accessed: 2025-07-30.

[12] Dart Team, Google. 2011. Dart Programming Language. https://dart.dev. Accessed: 2025-07-30.

[13] ECMA International. 2018. *ECMAScript 2025 Language Specification* (16.0 ed.). https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[14] A.M. Erosa and L.J. Hendren. 1994. Taming control flow: a structured approach to eliminating goto statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. 229–240. doi:10.1109/ICCL.1994.288377

[15] Ben Gamari, Andreas Klebinger, and Simon Peyton Jones. 1992. The Glasgow Haskell Compiler. https://www.haskell.org/ghc. Accessed: 2025-07-30.

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

[17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 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. doi:10.1145/3062341.3062363

[18] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 107–120. https://www.usenix.org/conference/atc19/presentation/jangda

[19] 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). http://www-sop.inria.fr/indes/fp/Bigloo/doc/r5rs.html

[20] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2021. *The Java Virtual Machine Specification, Java SE 17 Edition.* Oracle America, Inc. https://docs.oracle.com/javase/specs/jvms/se17/html/.

[21] Niko Mäkitalo, Victor Bankowski, Paulius Daubaris, Risto Mikkola, Oleg Beletski, and Tommi Mikkonen. 2021. Bringing WebAssembly up to speed with dynamic linking. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (Virtual Event, Republic of Korea) *(SAC '21)*. Association for Computing Machinery, New York, NY, USA, 1727–1735. doi:10.1145/3412841.3442045

[22] Olivier Melançon, Marc Feeley, and Manuel Serrano. 2024. Static Basic Block Versioning. In *38th European Conference on Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–27. doi:10.4230/LIPIcs.ECOOP.2024.28

[23] Thibaud Michaud and Thomas Lively. 2023. WebAssembly tail calls. https://v8.dev/blog/wasm-tail-call

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

[25] Norman Ramsey. 2022. Beyond Relooper: recursive translation of unstructured control flow to structured control flow (functional pearl). *Proc. ACM Program. Lang.* 6, ICFP, Article 90 (Aug. 2022), 22 pages. doi:10.1145/3547621

[26] John C. Reynolds. 1993. The Discoveries of Continuations. *LISP and Symbolic Computation* 6, 3-4 (1993), 233–247. doi:10.1007/BF01019459

[27] Andreas Rossberg. 2023. Mutually Iso-Recursive Subtyping. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 234 (Oct. 2023), 27 pages. doi:10.1145/3622809

[28] Andreas Rossberg, Deepti Gandluri, Luke Wagner, Alon Zakai, Dan Gohman, and Ben Smith. 2019. WebAssembly Core Specification. https://www.w3.org/TR/2019/REC-wasm-core-1-20191205/. W3C Recommendation, December 5, 2019.

[29] Prabhjot Sandhu, David Herrera, and Laurie Hendren. 2018. Sparse matrices on the web: Characterizing the performance and optimal format selection of sparse matrix-vector multiplication in JavaScript and WebAssembly. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes* (Linz, Austria) *(ManLang '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 13 pages. doi:10.1145/3237009.3237020

[30] Manuel Serrano. 1994. *Bigloo user's manual.* RT 0169. Inria-Rocquencourt, France. http://www.inria.fr/mimosa/fp/Bigloo

[31] Cheng Shao. 2018. Fibonacci compiles end-to-end – Haskell to WebAssembly via GHC. https://www.tweag.io/blog/2018-05-29-hello-asterius/

[32] Simon Peyton Jones and others. 1990. Haskell Language. https://www.haskell.org. Accessed: 2025-07-30.

[33] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. 1–10. doi:10.1109/CLEI53233.2021.9640153

[34] W. Richard Stevens. 1992. *Advanced Programming in the UNIX Environment.* Addison-Wesley.

[35] Anastasios Stotoglou and Theodore H. Kaskalis. 2023. Comparative Study of JavaScript and WebAssembly Derivatives in Browser Engines. In *2023 Intelligent Methods, Systems, and Applications (IMSA)*. 476–483. doi:10.1109/IMSA58542.2023.10217486

[36] Joshua Wenata Sunarto, Angelina Quincy, Fakhira Shafa Maheswari, Quesynovich Denis Al Hafizh, Melanie Gabriela Tjandrasubrata, and Mochammad Haldi Widianto. 2023. A Systematic Review of WebAssembly VS Javascript Performance Comparison. In *2023 International Conference on Information Management and Technology*

(ICIMTech). 241–246. doi:10.1109/ICIMTech59029.2023.10277917

[37] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. 2022. Leaps and bounds: Analyzing WebAssembly's performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 256–268. doi:10.1109/IISWC55918.2022.00030

[38] TIOBE Software BV. 2025. TIOBE Index. https://www.tiobe.com/tiobe-index/.

[39] Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The design and implementation of typed scheme. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 395–406. doi:10.1145/1328438.1328486

[40] Linus Wagner, Maximilian Mayer, Andrea Marino, Alireza Soldani Nezhad, Hugo Zwaan, and Ivano Malavolta. 2023. On the Energy Consumption and Performance of WebAssembly Binaries across Programming Languages and Runtimes in IoT. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering* (Oulu, Finland) *(EASE '23)*. Association for Computing Machinery, New York, NY, USA, 72–82. doi:10.1145/3593434.3593454

[41] Weihang Wang. 2021. Empowering Web Applications with WebAssembly: Are We There Yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1301–1305. doi:10.1109/ASE51524.2021.9678831

[42] Wasm_of_ocaml Team. [n. d.]. https://ocsigen.org/js_of_ocaml/latest/manual/wasm_overview. Accessed: 2025-08-01.

[43] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring the Edge up to Speed with A WebAssembly OS. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 353–360. doi:10.1109/CLOUD49709.2020.00056

[44] Andy Wingo. 2024. Scheme on WebAssembly: It is happening! https://icfp24.sigplan.org/details/scheme-2024-papers/6/Scheme-on-WebAssembly-It-is-happening-

[45] Xavier Leroy and contributors. 1996. The OCaml System. https://ocaml.org. Accessed: 2025-07-30.

[46] Yutian Yan, Tengfei Tu, Lijian Zhao, Yuchen Zhou, and Weihang Wang. 2021. Understanding the performance of webassembly applications. In *Proceedings of the 21st ACM Internet Measurement Conference* (Virtual Event) *(IMC '21)*. Association for Computing Machinery, New York, NY, USA, 533–549. doi:10.1145/3487552.3487827

[47] Dongjun Youn, Wonho Shin, Jaehyun Lee, Sukyoung Ryu, Joachim Breitner, Philippa Gardner, Sam Lindley, Matija Pretnar, Xiaojia Rao, Conrad Watt, and Andreas Rossberg. 2024. Bringing the WebAssembly Standard up to Speed with SpecTec. *Proc. ACM Program. Lang.* 8, PLDI, Article 210 (June 2024), 26 pages. doi:10.1145/3656440

[48] Yukihiro Matsumoto. 1995. Ruby Programming Language. https://www.ruby-lang.org. Accessed: 2025-07-30.

[49] Yixuan Zhang, Mugeng Liu, Haoyu Wang, Yun Ma, Gang Huang, and Xuanzhe Liu. 2025. Research on WebAssembly Runtimes: A Survey. *ACM Trans. Softw. Eng. Methodol.* (Jan. 2025). doi:10.1145/3714465 Just Accepted.

Received 2025-06-05; accepted 2025-07-28