JMM: Transpiling JavaScript to Rust

September 25, 2017

Abstract

By historical accident, JavaScript is the language used to program the modern web, despite its inherent performance limitations. WebAssembly is a new bytecode format that runs in the browser and promises significant performance improvements over JavaScript. However, current solutions for compiling JavaScript to WebAssembly are limited, leaving developers to port existing JavaScript codebases to WebAssembly. JavaScript Minus Minus (JMM) is a subset of JavaScript that compiles to WebAssembly or native code via Rust. This study defines JMM and presents its compiler. Benchmarks provide evidence of the potential for substantial performance improvements over existing JavaScript code, suggesting the potential of this technology to make the web faster.

1 Introduction

JavaScript has rapidly grown into one of the most popular programming languages [1][2] due to its position as the primary language for adding interactive functionality to websites [3]. Despite improvements in JavaScript engines, several design features of JavaScript such as a high garbage collection overhead and dynamic typing make it inherently slow [4]. Considering multi-million dollar cost of slightly longer page load times for internet businesses [5], the performance of JavaScript has significant economic implications [6]. Moreover, the internet has seen growing use in developing economies [7] where low performance devices are prevalent [8], including for critical infrastructure [9]. Thus, making web applications run more efficiently on all devices is of great social importance [10].

In the past decade, there several tools have been created for optimizing JavaScript. The most prominent of these is the Google Closure Compiler, which compiles Javascript into JavaScript, optimizing and minifying the code along the way with the aid of type annotations in comments [11]. There are also are hundreds of languages that compile to JavaScript [12]. Many languages that target JavaScript, such as Dart [13], Scala.js [14], and Elm [15], produce optimized JavaScript by taking advantage of unique language features such as strict typing [13] and restrictions such as immutability [16]. However, all of these tools still produce JavaScript, and thus suffer the runtime overhead of parsing, garbage collection, and weak types.

One proposed solution to this dilemma is WebAssembly, a proposal for a low level binary format that runs in all major browsers [17]. The main tool for generating WebAssembly, Emscripten, was originally designed to compile LLVM bytecode to asm.js [18], a predecessor of WebAssembly, but has now been repurposed to target WebAssembly [17]. LLVM languages such as C, C++, and Rust can be compiled to WebAssembly, outperforming equivalent JavaScript code due to the use of static types, lack of garbage collection, lack of runtime overhead, and LLVM's robust optimizations [17]. Given the robust community and corporate backing of WebAssembly [19], it seems likely that WebAssembly will continue to grow in popularity and adoption [20].

Unfortunately, the transition to WebAssembly will lead to several problems in the web development ecosystem. Because the LLVM ecosystem is currently centered around low level systems programming languages [18], it may be difficult for the hundreds of thousands of professional JavaScript developers [1] to transition to WebAssembly development. In addition to wasted human capital, the millions of dollars and human-hours invested into today's massive JavaScript codebases — such as npm, the largest package repository on Earth [21] — could go to waste. Moreover, the numerous languages that compile to JavaScript and have gained widespread adoption will each eventually need to rewrite their backends to target WebAssembly instead of JavaScript, incurring more development costs.

An efficient solution to this quandary would be a JavaScript to WebAssembly compiler. Unfortunately, completely supporting JavaScript while achieving superior performance is nearly impossible, mainly due to the lack of garbage collection in WebAssembly [17]. Differences in APIs and static typing would also pose difficulty. Although a full JavaScript engine such as V8 could theoretically be compiled to WebAssembly and used to run JavaScript, it would be much slower than if it were run natively.

There have been several attempts to compile JavaScript or a JavaScript-like language to WebAssembly. Speedy.js, ThinScript, TurboScript, and AssemblyScript are each based off of TypeScript's syntax, and make use of explicit, static types [22][23][24][25]. Unfortunately, each language currently lacks a mature feature set, lacking important scripting functionality such as strings, regular expressions, and objects [22][23][24][25]. Although these projects may be useful for developers hoping to transition toward WebAssembly development, the differences between them and vanilla JavaScript will likely make it difficult to port existing JavaScript codebases to these languages [25].

One reason that these languages have had difficulty implementing higher level features is that they target a low level intermediary such as LLVM bytecode [22] or Binaryen IR [23][24][25]. Before discussing this study's solution to the WebAssembly compilation problem, it is worth first considering other potential intermediaries between JavaScript and WebAssembly. One option is C or C++, to which there are several projects attempting to compile JavaScript [26] [27]. C/C++

could then be easily compiled to WebAssembly via the traditional toolchain. However, due to the memory management system of these languages, it is harder for a compiler to generate memory safe, error free code. Another approach that worth consideration is compiling JavaScript to .NET Common Intermediate Language via Jurassic [28], which could be compiled to LLVM IR via LLILC [29]. Unfortunately, LLILC not is mature enough [29] to effectively support this pipeline, and Jurassic is ill suited for the task as it is designed for integrating JavaScript into .NET applications [28].

This paper presents an alternative solution to the Javascript-to-WebAssembly compilation problem. Javascript Minus Minus (JMM) is a subset of JavaScript that compiles to WebAssembly through Rust. It supports all basic features of JavaScript such as arithmetic, logic, and strings, as well as more advanced features such as object oriented programming and arrays. Herein, JMM's syntax and semantics are introduced. The JMM compiler is also presented, including the design choices that went into it. Benchmarks are used to demonstrate JMM's performance and the ease with which JavaScript can be ported to JMM. Finally, a roadmap for bringing JMM to production readiness is outlined.

2 JMM: The Language

JMM is a subset of EcmaScript 5.1 [30], a widely supported specification of JavaScript [31]. For more modern versions of JavaScript such as EcmaScript 6, developers use tools such as Babel to compile down to EcmaScript 5 for production [32]. JMM can become the next phase in the compilation toolchain, converting EcmaScript 5 to WebAssembly.

The main differences between JMM and idiomatic JavaScript are its memory management model and its type system. Additional differences include its Object Oriented Programming (OOP) patterns and the restricted semantics of its statements. JMM includes a limited subset of the JavaScript API, but larger portions could be implemented in the future.

Valid JMM is semantically equivalent and valid JavaScript, but not all JavaScript is valid JMM.

JMM has been designed, however, for the ease of porting of large swathes of existing JavaScript codebases.

2.1 Memory Management

Unlike JavaScript [33], WebAssembly lacks a garbage collector [17]. Rather than implementing garbage collection, which would create an expensive runtime overhead, Rust uses a set of compile time rules for references that guarantee memory safety without garbage collection [34].

JMM borrows from Rust in this regard, making use of Rust's memory management rules. JavaScript's immutable types — numbers, booleans, and strings — can be used as they normally are in JavaScript [35]. However, mutable types, namely arrays and objects, follow Rust's borrow checking semantics. There may only be one mutable reference per scope for these types.

The borrow checking rule is somewhat restrictive, and one workaround is cloning. JMM makes use of JavaScript semantics, so cloning an object `x` is done by calling `JSON.parse(JSON.stringify(x))`. The compiler treats these function calls as a deep copy operation.

2.2 Types

Another difference between JavaScript and JMM is the type system. JavaScript is a dynamically typed language, in which the same variable can hold values of multiple types within the same scope [35]. JavaScript allows this semantic because it is an interpreted or JIT-compiled language [35].

However, JMM is a compiled, statically typed language. All variables' types in JMM must be either declared or inferrable at compile time. Types may be declared using JSDoc comments, a popular standard for specifying the types of JavaScript variables [36].

In addition, JMM does not permit `null` or `undefined` values. These have been long criticized as a dangerous language feature, the source of many bugs in programs written in JavaScript and other languages that allow equivalent values [37][34]. In lieu of these values, JMM programs can use objects to simulate *option types*.

JMM does not support object literals. It supports string, number, boolean, and array literals. Objects can be constructed by calling constructor functions.

2.3 Object Oriented Programming

JavaScript uses a prototypal inheritance, in which objects descend from objects rather than classes [38]. While this makes for convenient scripting, it also makes compile time analysis difficult.

JMM supports a semantic subset of JavaScript's OOP facilities that is effectively a class based OOP system. The syntax for each of these constructs was selected based on the most common JavaScript OOP patterns [39].

Constructors and Instantiation JMM considers any function that is called with a `new` expression a constructor function. Constructors are like any other function in JMM, except that they may use `this` to set properties of the object. An object's fields are inferred based on locations in the constructor in which `this.field` is set to a particular value.

Instance Methods Instance methods have access to `this`, which refers to an instance of the class.

JMM looks for instance methods in two of the most common JavaScript OOP patterns. The first is the form

```
Class.prototype = {method:  function(...){...}, ...}
```

where `Class` is the name of a class and the properties and `method` is an instance method.

The second OOP pattern is of the form

```
Class.prototype.method = function(...){...}; ...
```

for each of the class's methods.

Use of Objects A limited set of object expressions are supported in JMM. Specifically, properties not specified in the constructor may not also be attached to an object afterwards. Otherwise, objects may be used as in JavaScript, as long as usage follows Rust's borrowing rules.

2.4 Other Semantics

2.4.1 Functions

All functions must be either type-inferable or specify types of their parameters and return value using JS Doc syntax.

Like Rust [34], JMM makes a distinction between functions and function expressions. Functions are specified in the form:

```
function name(param1, param2, ...){...}
```

Functions may not access dynamic global scope, and may not be passed as arguments, used as object properties or array elements, or otherwise treated as variables. Functions may use recursion.

Function expressions are not yet supported, except as instance methods per Section 2.3.

2.4.2 Control Flow

Control flow in JMM is largely the same as in JavaScript, supporting the following statements equivalently: `if-else`, `while`, `do-while`, and `for(-;-;-)`.

`for-in` and `for-of` loops are not supported. Additionally, `switch` statements only allow for limited semantics; `break` statements are only allowed at the top level of the body of a `case`. These restricted semantics for `switch` encompass typical use, as nested `break` statements can lead to unpredictable behavior and are considered bad practice [40]. In all other constructs, `break` statements function as in JavaScript.

2.4.3 Variables

Like JavaScript, all variables are mutable. Valid JMM identifiers are any valid JavaScript identifiers that do not use the `$` symbol. Moreover, JMM does not support variable hoisting as is performed in JavaScript [41]. However, making use of hoisting is generally considered a bad practice in the JavaScript community [41]. Assignment to variables follows the standard syntax but assignment expressions are not permitted.

2.4.4 Operators

All JavaScript logical, arithmetic, and bitwise operators are supported except for the zero-fill right shift operator (>>>). Member access on arrays is also supported.

2.5 Supported APIs

Currently JMM supports only a small subset of the JavaScript API. It supports `Math.sqrt`, `process.argv`, and `console.log`; the `Float64Array` constructor is also included.

3 JMM: The Compiler

This section, the JMM compiler. Its source code is available at `https://goo.gl/EF9VzU`.

3.1 Rust

Rust is a new systems programming language created by Mozilla [42]. The JMM compiler compiles JMM to Rust; the Rust compiler then produces either WebAssembly or native binaries.

Rust was chosen as an intermediary between JavaScript and WebAssembly for several reasons. It guarantees memory safety without the overhead of garbage collection through compile time analysis and a set of memory management rules. This guarantees memory safety and reduces the possibility of runtime errors [34]. Prevention of memory leaks makes Rust an attractive compiler target for JMM and helps ensure the correctness of the compiler.

Additionally, Rust has a robust and well tested toolchain that already supports compilation to WebAssembly through a wrapper over Emscripten [43]. Rust compiles to LLVM bytecode which can then target numerous platforms. This approach allows JMM to take advantage of decades of work in LLVM optimizations, such as dead code elimination and link time optimization [44]. As LLVM grows in popularity and its optimizations improve, JMM code will become faster. Likewise, the advent of a middle-level intermediate representation may make Rust, and in turn JMM, significantly faster over time [45].

Another benefit of Rust is a growing ecosystem of libraries from which JMM can draw in order to implement features such as JSON [46], regular expressions [47], asynchronous [48] and parallel programming [49], as well as other JavaScript APIs.

Lastly, Rust is structurally similar to JavaScript, as both languages descend from C. With the exception of Object Oriented Programming, there are striking structural similarities between the two languages. This is what makes transpilation feasible.

3.2 Architectural Overview

JMM compilation architecture consists of parsing, "OOP extraction," and recursive, depth-first, abstract syntax tree (AST) traversal. The generated Rust code is then appended onto a library of rust code which provides various JavaScript APIs.

The JMM takes advantage of the robust ecosystem of JavaScript parsing and analysis tools. The two preprocessing libraries used are Acorn, a fast EcmaScript parser, which generates an AST from JavaScript code [50], and Tern, a type inference engine that is compatible with Acorn's AST format [51].

3.3 Classes

JavaScript makes use of prototypal inheritance and Rust lacks any form of inheritance. This semantic difference makes direct translation between the two languages impossible. Instead of supporting the full range of JavaScript OOP, JMM uses a small subset that simulates a class-based OOP model, without inheritance, which can be translated into Rust `structs` and `impl` blocks.

To detect classes, the JMM compiler looks for `new` expressions, then filters the resulting list of class names for classes defined by the JavaScript API. The constructor is the function of the same name as the class. The compiler infers the fields of a class by looking in the constructor for member expressions of the form `this.property`. For each class, the compiler looks for assignments following the OOP patterns mentioned in Section 2.3 to detect instance methods.

Classes are converted into Rust `structs`, where the fields are defined by the inferred fields and their types by their names. Instance methods are transformed by changing `this` expressions to `self`, where `self` is a mutable reference parameter prepended to the parameter list. These methods are then transpiled to Rust and placed inside an `impl` block for the struct. The constructor is also transpiled, with `this` initialized to a struct with dummy values at the beginning of the body and returned by the function in order to be valid Rust. `new` expressions are converted to the form `Class::new(...)`.

Lastly, the compiler removes the constructor and instance methods from the AST, allowing the transpilation process to continue.

3.4 Rust Library

Although the compiler attempts to compile JavaScript code directly into Rust, the generated Rust often depends on JavaScript APIs. Moreover, some expressions such as logical operators require runtime type determination. Thus, a Rust library is included to provide these capabilities.

3.4.1 Identifier Prefixing

As a convention, the compiler mangles JavaScript identifiers during Rust code generation by prefixing them with `__js__`. This convention, which violates Rust's naming conventions but is syntactically permissible [52], prevents unintentional collisions between JMM generated Rust and other Rust code. It also allows the exposure of methods to JavaScript, such as `Math.sqrt` to be explicitly indicated, by prefixing: `__Math__.__sqrt__`.

3.4.2 Traits

The following traits are declared and implemented for the Rust types into which JavaScript numbers, strings, and booleans are converted so that prefixed methods can be exposed to JavaScript and to provide the compiler with runtime information:

ToString Provides prefixed `toString` method, a string representation of JavaScript primitives and arrays.

ToNum Provides an unprefixed method `to_num` which is used to coerce types into numbers. This is used by the compiler to perform arithmetic operations between differently typed arguments.

ToBool Provides an unprefixed method `is_truthy` for use in logical operations between non-boolean types.

TypeOf Provides an unprefixed method `type_of` which returns a string representation of the type. This is needed to support JavaScript's `typeof` expression, as Rust does not support reflection at runtime.

3.4.3 Arrays

In order to facilitate JavaScript's Array APIs, a `__js__Array` type is provided as a wrapper over Rust's `Vec` type. This provides the `push` method and `length` property. It also implements immutable and mutable indexing, and a `new` method for array construction.

3.4.4 Other Utilities

Functions for logical operators are provided in the rust library. Logical operators are compiled into rust function calls by the compiler, as logical operators require runtime type information.

Several common global functions and objects such as `console.log`, `process.argv`, `Math.sqrt`, and `Float64Array` are provided by the Rust library as thin wrappers over the corresponding Rust libraries.

3.5 Data Types and Operations

JavaScript types do not directly correspond with Rust types. The primitive types are directly compiled into their counterparts: JavaScript's `Number` type, a double precision floating point number [35], is converted into Rust's `f64`; its `Boolean` type is converted into Rust's `bool`; and its `String` type is converted into Rust's owned string type, `String` [34].

Although JavaScript arrays are objects, they are converted into Rust `Vec`s, which support much of the same functionality. JavaScript objects, as mentioned in section 3.3, are converted into Rust `struct`s.

3.5.1 Operators

Most of JavaScript's operators have Rust counterparts, but Rust operators are strongly typed [34] whereas JavaScript generously type coerces operands [35]. The JMM compiler attempts to account for this behavior while minimizing runtime overhead.

Comparison operations are directly compiled if the types of each operand are the same. If not, the compiler attempts to coerce the operands into numbers. Arithmetic operations work similarly, with the exception of addition, which is converted into string concatenation if either operand is not coercible into a number. Logical operations perform type coercion into booleans through the `is_truthy` method. Bitwise operations convert their operands into numbers; in order to match

JavaScript semantics, the operands are then casted from 64-bit floating point numbers to 64-bit integers, the operation is performed, and the operands are casted back into floats.

3.6 Syntactic Constructs

3.6.1 Functions

Declaration With Tern's type signature inference, JavaScript functions are compiled into Rust functions with the corresponding type signature. The function's block is compiled using as any other JMM block is. As long as the function's return value is of a single type, `return` statements in JavaScript are semantically equivalent to those in Rust, so they too are transpiled directly.

Functions Calls Function calls are transpiled by inferring the type of reference. For immutable types, namely booleans and numbers, the argument is passed by value. Although strings are immutable in JavaScript, their Rust counterpart is mutable, so a clone is performed. All other argument types are mutable and are thus transpiled into mutable references.

3.6.2 Control Flow

Control flow statements are almost identical in Rust and JavaScript, and so they can be transpiled at the statement level.

Conditional Statements and the Conditional Operator Conditional statements — `if-else` — are semantically equivalent in Rust and JavaScript and thus are treated equivalently. Although Rust has no conditional operator like JavaScript's `-?-:-`, its `if-else` statements are expressions [34], allowing transpilation of conditional operators.

Loops JavaScript's `while` loop behaves equivalently to Rust's. Rust does not have a `do-while` loop, so these loops are transpiled by placing one iteration outside of a while loop, thus providing the first iteration. Rust does support `for` loops, but they function like `for-of` loops instead of the traditional C-style `for(-;-;-)` loops. The initialization of these loops is hoisted out of the loop, the condition is used as the condition of a `while` loop, and the update is placed at the end of the block. Both `do-while` and `for` loops are nested in a block in the generated Rust to comply with JavaScript's scoping semantics.

Switch JavaScript's `switch` statement uses semantics inherited from C and permits a complex set of behavior [40]. Unfortunately, Rust's `match`, a roughly analogous statement, does not allow much of the behavior supported by JavaScript, such as nesting `break` statements or fall-through cases [34]. Hence, `switch` statements are transpiled into nested `if-else` blocks. The compiler looks for top-level `break` statements to infer the logical structure that should be generated.

Variable Declaration and Assignment As all JavaScript variables in EcmaScript 5 are declared via the `var` statement and can be reassigned later [30]. Thus, variable declarations are all declared with `let mut` in the generated Rust code. Assignment to variables is syntactically the same in both languages [34][30], assuming it is not treated in the JavaScript code as an expression; thus, assignment statements can be trivially transpiled to Rust.

4 Benchmarks

In order to determine the efficacy of JMM at improving JavaScript performance, benchmarks were conducted comparing JavaScript code to JMM, with C++ as a performance baseline.

4.1 Materials and Methods

Three programs from the Computer Language Benchmarks Game [53] were ported from JavaScript to JMM.

The performance was measured by running each program 10 times on an Amazon Elastic Cloud Compute t2.micro instance with 1 GiB RAM and 1 vCPU [54] running Ubuntu Server 16.04. Tests were run using Node 8.5.0, which uses version 5.8 of the V8 JavaScript runtime [55]. GCC 7.1.1

at its highest optimization settings was used to compile C++ code, as describe by the Computer Language Benchmarks Game specifications [53].

In addition to comparing the JavaScript and compiler generated WebAssembly, two additional versions of the same program were compared. To measure performance differences between WebAssembly itself and native code, JMM code was also compiled to a native binary. Lastly, C++ compiled to native binary was used as a baseline for optimal performance on each benchmark. Because WebAssembly does not yet support concurrency [17], the fastest non-parallel implementations of C++ and JavaScript code were used for the benchmarks.

The following programs were selected:

Fannkuch A mathematical benchmark used to measure numerical operation performance [56].

N-Body A physics simulation that uses objects and numerical integration.

Spectral Norm A numerical computation on matrices with heavy memory use.

Porting each benchmark from JavaScript to JMM required only trivial modifications. The Fannkuch benchmark's sole modification was type annotations for three variables which Tern was unable to infer. The Spectral Norm benchmark also required only the addition of type annotations, in this case specifying the type signatures of functions. The N-Body benchmark required more modification: constant inlining, due to the inability of functions to access global scope; several type annotations and clone operations were added to comply with Rust's type checker and borrow checker, respectively.

The source code for each of the benchmarks and the shell script used to perform the benchmarks is available with the compiler's source code.

4.2 Results

Results from the benchmark are shown in Table 1 and Figure 1.

	Fannkuch	**N-Body**	**Spectral Norm**
JS	$48.14 \pm 0.05\%$	$327.7 \pm 0.69\%$	$6.709 \pm 0.12\%$
JMM/WASM	$107.7 \pm 0.01\%$	$21.51 \pm 0.10\%$	$11.64 \pm 0.07\%$
JMM/BIN	$77.04 \pm 0.02\%$	$7.996 \pm 0.03\%$	$6.638 \pm 0.11\%$
C++	$16.69 \pm 0.11\%$	$4.807 \pm 0.17\%$	$6.683 \pm 0.12\%$

Table 1: Mean and Standard Deviation of Benchmarks

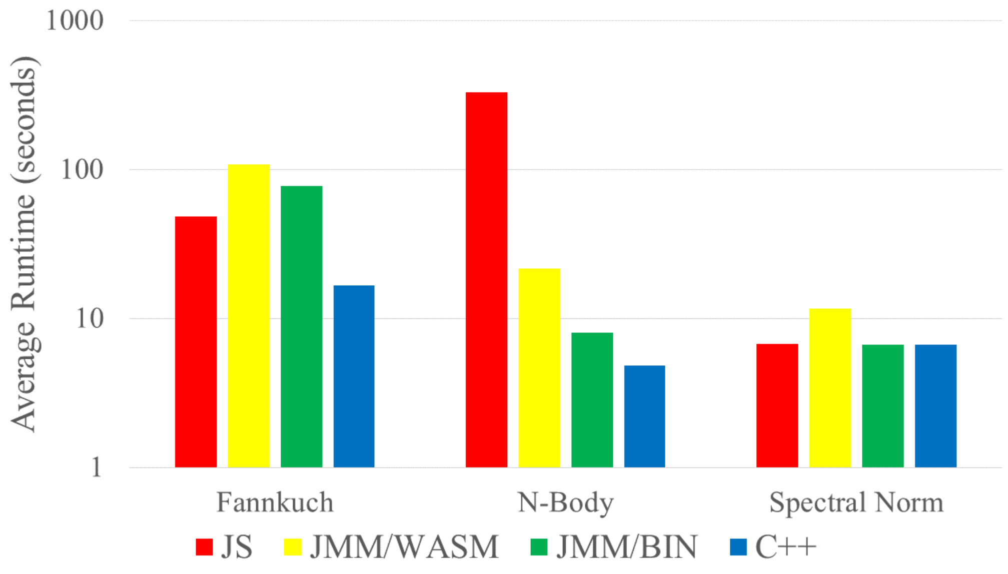

Figure 1: Performance of Benchmarks vs. Language and Compiler Target

Using one-way analysis of variance [57], it was computed for each benchmark that $p < 0.001$, a statistically significant result.

4.3 Discussion

In each benchmark, C++ had superior or equal performance to JavaScript and JMM code; the margins in the Fannkuch and N-Body benchmarks were substantial. There is clearly much work ahead before JavaScript matches this performance.

In addition, JMM code compiled to WebAssembly was consistently slower than the same code compiled to native binary. This is likely due to inherent limitations in the current experimental WebAssembly virtual machines and the lack of platform specific optimizations for WebAssembly compilation. Over time, the gap between native and WebAssembly may shrink as WebAssembly improves [58]. Thus, it is worth comparing performance of JavaScript on V8, a popular JavaScript engine, versus JMM compiled to native, which serves as the performance upper bound of JMM compiled to WebAssembly.

In the Fannkuch benchmark, JavaScript outperforms JMM by nearly 50%. This benchmark makes use of simple array and numerical operations, which V8 may be able to optimize more effectively. The JMM compiler wraps the vector into the `__js__Array` struct, which may be the source of JMM's inferior performance.

For the Spectral Norm benchmark, the performance of all targets except for WebAssembly is equal. That JavaScript is able to match native C++ performance is a testament to the substantial work in browser engineering. It is equally impressive that JMM compiled to native code is able to match the performance of C++. However, as all platforms were equally performant, this benchmark provides little further insight.

Lastly, the N-Body benchmark show substantial performance gains from JMM over JavaScript; native JMM was, on average, 40 times faster than JavaScript! JMM compiled to WebAssembly also outperformed JavaScript by a significant margin. It is unclear what accounts for this huge disparity; it seems improbable to be solely attributable to static typing and manual memory management. Further research is required.

5 Conclusion

JMM shows much promise as a solution for the compilation of JavaScript to WebAssembly. This study has demonstrated that small modifications to existing JavaScript, allowing for compilation to WebAssembly or native code, can lead to substantial performance gains.

Further work on JMM is required to expand the compilable subset of JavaScript enough to include most real world code. The largest difference between JMM and JavaScript is the use of static types; but this difference need not change. Many JavaScript programmers already use type checkers or language supersets that include type checking, both of which prevent bugs [59].

Moreover, the use of type inference by the JMM compiler enables most well typed JavaScript code to be used with little modification.

Future improvements of JMM must focus on a few critical areas. First, the current memory management model is too restrictive for many existing JavaScript programs to be used unmodified. It may also lead to runtime overhead, as passing the borrow checker sometimes requires unnecessary deep copies. The solution may be using Rust's built-in reference counting system [34] with a cycle detector to prevent memory leaks. Using a lightweight garbage collector is also worth investigation.

Other JavaScript behavior that JMM does not currently support include JavaScript's objects, which have prototypal inheritance and also serve as untyped hash maps. Another area that must be supported is support for lexical closures, especially for functional expressions. Although these features are not strictly necessary for most tasks, they are quite prevalent and including them would make porting from JavaScript much easier.

Lastly, in order for JMM to truly encompass a usable subset of JavaScript, a much larger set of the standard JavaScript API must be supported, including regular expressions, string and array methods, and mathematics. Tooling that allows for two way interoperation between JavaScript and JMM-generated WebAssembly would also be required for practical use of JMM in the browser.

In addition to new features, performance improvements should certainly be considered. Many improvements will come as WebAssembly VMs mature [58] and as the Rust compiler adds more optimizations [45]. However, JavaScript specific optimizations should be explored. Pairing the JMM compiler with a JavaScript-to-JavaScript optimizing compiler such as the Google Closure Compiler [11] or PrePack [60] also merits consideration.

JavaScript to WebAssembly compilation shows promise as a source for performance improvements on the web. This study has shown the potential of JMM to make the web faster and thus more accessible for all of its users. The performance of JMM binaries also suggests that JMM may be useful for using JavaScript in embedded systems without the memory overhead of traditional JavaScript engines. Lastly, JMM should be considered as a target for languages that compile to

JavaScript; by compiling to JMM, they can achieve significant performance improvements and expedite the transition to WebAssembly.

References

[1] M. Byrne, "Stack overflow survey: Javascript is the most popular programming language," 2016. Available at https://motherboard.vice.com/en_us/article/jpgpkk/stack-overflow-survey-javascript-is-the-most-popular-programming-language.

[2] S. Cass, "The 2017 top programming languages," 2017. Available at https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages.

[3] "What is javascript?," 2017. Available at https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps/What_is_JavaScript.

[4] D. Crawford, "Why mobile web apps are slow," 2013. Available at http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/.

[5] K. Eaton, "How one second could cost amazon 1.6 billion in sales," 2012. Available at https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales.

[6] S. Souders, "High-performance web sites," 2008. Available at https://cacm.acm.org/magazines/2008/12/3358-high-performance-web-sites/fulltext.

[7] J. Poushter, "http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/," 2016. Available at http://www.pewglobal.org/2016/02/22/smartphone-ownership-and-internet-usage-continues-to-climb-in-emerging-economies/.

[8] J. Poushter, J. Bell, and R. Oates, "Internet seen as positive influence on education but negative on morality in emerging and developing nations," 2015. Available at http://assets.pewresearch.org/wp-content/uploads/sites/2/2015/03/Pew-Research-Center-Technology-Report-FINAL-March-19-20151.pdf.

[9] F. Dews, “How the internet and data help the developing world,” 2014. Available at https://www.brookings.edu/blog/brookings-now/2014/02/06/how-the-internet-and-data-help-the-developing-world/.

[10] “Advancing internet access in developing countries can help achieve sustainable economies un official,” 2012. Available at http://www.un.org/apps/news/story.asp?NewsID=43459.

[11] G. Richards, S. Lebresne, B. Burg, and J. Vitek, “An analysis of the dynamic behavior of javascript programs,” in *ACM Sigplan Notices*, vol. 45, pp. 1–12, ACM, 2010.

[12] “List of languages that compile to js.” Available at https://github.com/jashkenas/coffeescript/wiki/List-of-languages-that-compile-to-JS.

[13] S. Mohanty, S. R. Dey, *et al.*, “Dart evolved for web-a comparative study with javascript,” in *IJCA Proceedings on International Conference on Emergent Trends in Computing and Communication (ETCC-2014)*, no. 1, pp. 73–77, Foundation of Computer Science (FCS), 2014.

[14] “Scala.js.” Available at http://www.scala-js.org/.

[15] E. Czaplicki, “Elm: Concurrent frp for functional guis,” *Senior thesis, Harvard University*, 2012.

[16] E. Czaplicki, “Blazing fast html,” Aug. 2016. Available at http://elm-lang.org/blog/blazing-fast-html-round-two.

[17] D. L. S. B. L. T. D. G. L. W. A. Z. M. H. J. B. Andreas Haas, Andreas Rossberg, “Bringing the web up to speed with webassembly.” Available at https://github.com/WebAssembly/spec/blob/master/papers/pldi2017.pdf, June 2017.

[18] A. Zakai, “Emscripten: An llvm-to-javascript compiler,” in *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA ’11, (New York, NY, USA), pp. 301–312, ACM, 2011.

[19] R. Chartrand, “The top rising javascript trends to watch in 2017.” Web, Dec. 2016. Available at https://medium.com/commit-push/the-top-rising-javascript-trends-to-watch-in-2017-86d8e87db3b3.

[20] D. Bryant, “Why webassembly is a game changer for the webŁŁand a source of pride for mozilla and firefox.” Web, Mar. Available at https://medium.com/mozilla-tech/why-webassembly-is-a-game-changer-for-the-web-and-a-source-of-pride-for-mozilla-an

[21] P. Brown, “State of the union: npm,” Jan. 2017. Available at https://www.linux.com/news/event/Nodejs/2016/state-union-npm.

[22] M. Reiser, “Speedy.js language reference,” 2017. Available at https://github.com/MichaReiser/speedy.js/wiki/Speedy.js-Language-Reference.

[23] E. Wallace, “Thinscript,” 2016. Available at https://evanw.github.io/thinscript/.

[24] “Turboscript,” 2017. Available at https://github.com/01alchemist/TurboScript.

[25] “Assemblyscript,” 2017. Available at https://github.com/AssemblyScript/assemblyscript.

[26] F. Santos, “js2cpp,” 2017. Available at https://github.com/fabiosantoscode/js2cpp.

[27] A. Markeev, “Javascript/typescript to c transpiler,” 2017. Available at https://github.com/andrei-markeev/ts2c.

[28] P. Bartrum, “Jurassic,” 2017. Available at https://github.com/fabiosantoscode/js2cpp.

[29] “Llilc,” 2016. Available at https://github.com/dotnet/llilc.

[30] E. ECMAScript, E. C. M. Association, *et al.*, “Ecmascript language specification,” 2011.

[31] J. Zaytsev, “Ecmascript compatibility table.” Web. Available at `https://kangax.github.io/compat-table/es5/`.

[32] “Babel.” Available at `https://babeljs.io/`.

[33] “Memory management.” Web, 2017. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management`.

[34] S. Klabnik and C. Nichols, “The rust programming language.” Available at `https://lise-henry.github.io/books/trpl2.pdf`.

[35] “Javascript data types and data structures,” 2017. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures`.

[36] “Jsdoc,” 2017. Available at `http://usejsdoc.org/`.

[37] T. Hoare, “Null references: The billion dollar mistake,” *Presentation at QCon London*, vol. 298, 2009.

[38] “Inheritance and the prototype chain,” 2017. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain`.

[39] J. Dugan, “Object oriented javascript pattern comparison,” Feb. 2015. Available at `https://john-dugan.com/object-oriented-javascript-pattern-comparison/`.

[40] T. Motto, “Replacing switch statements with object literals,” July 2014. Available at `https://toddmotto.com/deprecating-the-switch-statement-for-object-literals/#problems-with-switch`.

[41] “var,” 2017. Available at `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/var`.

[42] “The rust programming language.” Available at `https://www.rust-lang.org/en-US/`.

[43] S. Rubanov, "Compiling rust to webassembly guide," Jan. 2017. Available at `https://hackernoon.com/compiling-rust-to-webassembly-guide-411066a69fde`.

[44] "Llvms analysis and transform passes," Sept. 2017. Available at `https://llvm.org/docs/Passes.html`.

[45] N. Matsakis, "Introducing mir," Apr. 2016. Available at `https://blog.rust-lang.org/2016/04/19/MIR.html`.

[46] "Serde," 2017. Available at `https://serde.rs/`.

[47] "Crate regex," 2015. Available at `https://doc.rust-lang.org/regex/regex/index.html`.

[48] "Mio," Carl Lerche. Available at `https://github.com/carllerche/mio`.

[49] N. Matsakis, "Rayon," 2017. Available at `https://github.com/nikomatsakis/rayon`.

[50] "Acorn." Available at `https://github.com/ternjs/acorn`.

[51] "Tern: Intelligent javascript tooling." Available at `http://ternjs.net/`.

[52] "The reference," 2017. Available at `https://doc.rust-lang.org/reference/`.

[53] I. Guoy, "The computer language benchmarks game." Available at `http://benchmarksgame.alioth.debian.org/`.

[54] "Amazon ec2 instance types." Available at `https://aws.amazon.com/ec2/instance-types/`.

[55] J. M. Snell and C. Ihrig, "Node v8.0.0 (current)," May 2017. Available at `https://nodejs.org/en/blog/release/v8.0.0/`.

[56] K. Anderson and D. Rettig, "Performing lisp analysis of the fannkuch benchmark."

[57] D. M. Lane, *Online Statistics Education: An Interactive Multimedia Course of Study*, ch. XV. Available at `http://onlinestatbook.com/2/analysis_of_variance/ANOVA.html`.

[58] L. Clark, "Where is webassembly now and whats next?," Feb. 2017. Available at `https://hacks.mozilla.org/2017/02/where-is-webassembly-now-and-whats-next/`.

[59] Z. Gao, C. Bird, and E. T. Barr, "To type or not to type: quantifying detectable bugs in javascript," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 758–769, IEEE Press, 2017.

[60] "Prepack," 2017. Available at `https://prepack.io/`.